Documentation

Exported from JBoss Community Documentation Editor at 2018-03-05 04:05:10 EST

Table of Contents

Welcome to the WildFly Documentation. The documentation for WildFly is split into two categories:

- *Administrator Guides* for those wanting to understand how to install and configure the server
- *Developer Guides* for those wanting to understand how to develop applications for the server

There is also the WildFly Model Reference that provides information about all subsystem configuration options generated directly from the management model.

1 Administrator Guides

- The Getting Started Guide shows you how to install and start the server, how to configure logging, how to deploy an application, how to deploy a datasource, and how to get started using the command line interface and web management interface
- The Admin Guide provides detailed information on using the CLI and Web Management interface, how to use the domain configuration, and shows you how to configure key subsystems
- The High Availability Guide shows you how to create a cluster, how configure the web container and EJB container for clustering, and shows you how to configure load balancing and failover

2 Developer Guides

- The Getting Started Developing Applications Guide shows you how to build Java EE applications and deploy them to WildFly. The guide starts by showing you the simplest *helloworld* application using just Servlet and CDI, and then adds in JSF, persistence and transactions, EJB, Bean Validation, RESTful web services and more. You'll also discover how to deploy an OSGi bundle to WildFly. Finally, you'll get the opportunity to create your own skeleton project. Each tutorial is accompanied by a quickstart, which contains the source code, deployment descriptors and a Maven based build.
- The Developer Guide (*in progress*) takes you through every deployment descriptor and every annotation offered by WildFly.
- The JavaEE 6 Tutorial (*in progress*) builds on what you learnt in the Getting Started Developing Applications Guide, and shows you how to build a complex application using Java EE and portable extensions.
- The Extending WildFly guide walks you through creating a new WildFly subsystem extension, in order to add more functionality to WildFly, and shows how to test it before plugging it into WildFly.

3 Quickstarts

WildFly comes with a number of quickstarts, examples which introduce to a particular technology or feature of the application server. The Contributing a Quickstart section of the documentation details the available quickstarts

4 More Resources

- Glossary
- WildFly project page
- WildFly issue tracker
- WildFly user forum
- WildFly wiki
- WildFly source

5 Admin Guide

5.1 Target audience

This document is a guide to the setup, administration, and configuration of WildFly.

5.1.1 Prerequisites

Before continuing, you should know how to download, install and run WildFly. For more information on these steps, refer here: Getting Started Guide.

5.1.2 Examples in this guide

The examples in this guide are largely expressed as XML configuration file excerpts, or by using a representation of the de-typed management model.

5.2 Management clients

WildFly offers three different approaches to configure and manage servers: a web interface, a command line client and a set of XML configuration files. Regardless of the approach you choose, the configuration is always synchronized across the different views and finally persisted to the XML files.

5.2.1 Web Management Interface

The web interface is a GWT application that uses the HTTP management API to configure a management domain or standalone server.

HTTP Management Endpoint

The HTTP API endpoint is the entry point for management clients that rely on the HTTP protocol to integrate with the management layer. It uses a JSON encoded protocol and a de-typed, RPC style API to describe and execute management operations against a managed domain or standalone server. It's used by the web console, but offers integration capabilities for a wide range of other clients too.

The HTTP API endpoint is co-located with either the domain controller or a standalone server. By default, it runs on port 9990:

```
<management-interfaces>
 [...]
 <http-interface security-realm="ManagementRealm">
  <socket-binding http="management-http"/>
 </http-interface>
<management-interfaces>
```

(See standalone/configuration/standalone.xml or domain/configuration/host.xml)

The HTTP API Endpoint serves two different contexts. One for executing management operations and another one that allows you to access the web interface:

- Domain API: http://<host>:9990/management
- Web Console: http://<host>:9990/console

Accessing the web console

The web console is served through the same port as the HTTP management API. It can be accessed by pointing your browser to:

- http://<host>:9990/console

Default URL

By default the web interface can be accessed here: http://localhost:9990/console.

Default HTTP Management Interface Security

WildFly is distributed secured by default. The default security mechanism is username / password based making use of HTTP Digest for the authentication process.

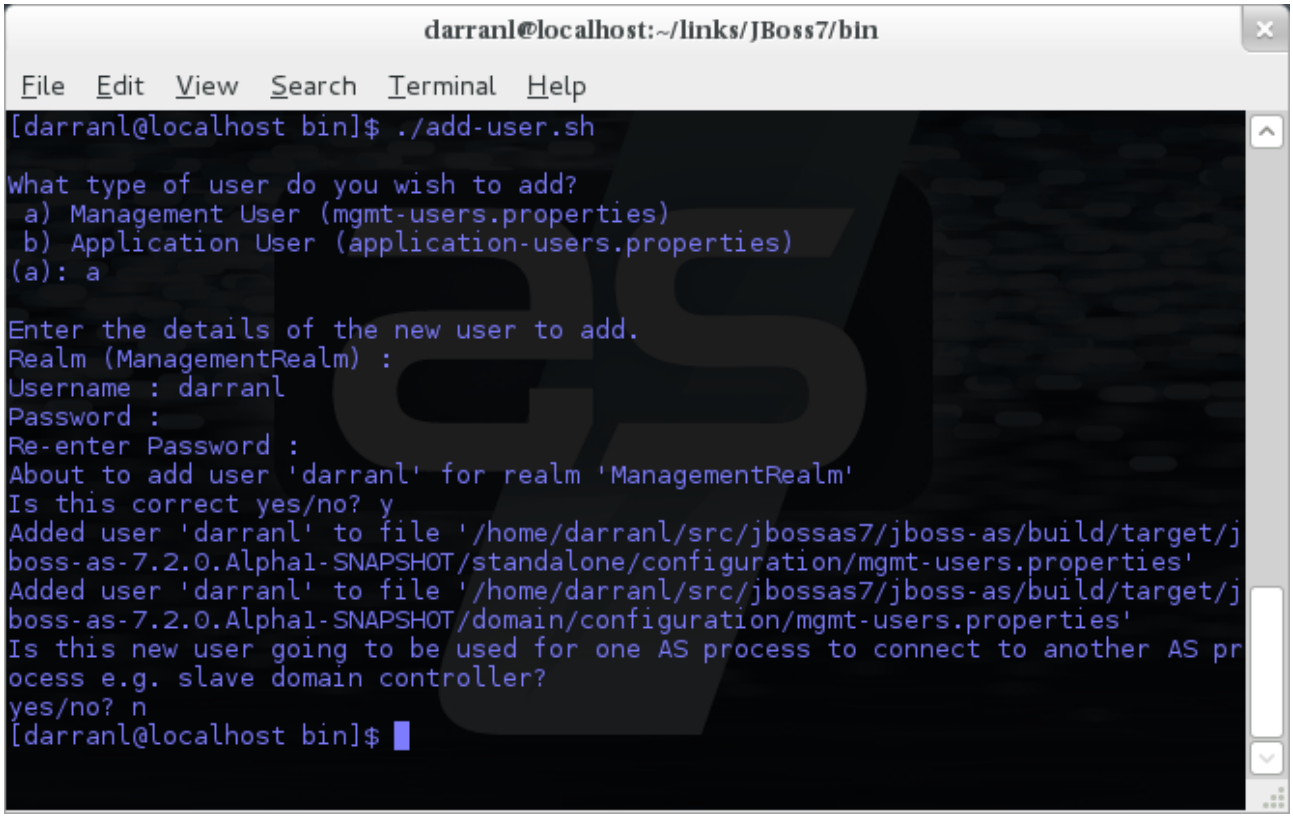
The reason for securing the server by default is so that if the management interfaces are accidentally exposed on a public IP address authentication is required to connect - for this reason there is no default user in the distribution.

If you attempt to connect to the admin console before you have added a user to the server you will be presented with the following screen.

The user are stored in a properties file called mgmt-users.properties under standalone/configuration and domain/configuration depending on the running mode of the server, these files contain the users username along with a pre-prepared hash of the username along with the name of the realm and the users password.

Although the properties files do not contain the plain text passwords they should still be guarded as the pre-prepared hashes could be used to gain access to any server with the same realm if the same user has used the same password.

To manipulate the files and add users we provide a utility add-user.sh and add-user.bat to add the users and generate the hashes, to add a user you should execute the script and follow the guided process.



The full details of the add-user utility are described later but for the purpose of accessing the management interface you need to enter the following values: -

- Type of user - This will be a 'Management User' to selection option a.
- Realm - This MUST match the realm name used in the configuration so unless you have changed the configuration to use a different realm name leave this set as 'ManagementRealm'.
- Username - The username of the user you are adding.
- Password - The users password.

Provided the validation passes you will then be asked to confirm you want to add the user and the properties files will be updated.

For the final question, as this is a user that is going to be accessing the admin console just answer 'n' - this option will be described later for adding slave host controllers that authenticate against a master domain controller but that is a later topic.

Updates to the properties file are picked up in real time so either click 'Try Again' on the error page that was displayed in the browser or navigate to the console again and you should then be prompted to enter the username and password to connect to the server.

5.2.2 Command Line Interface

The Command Line Interface (CLI) is a management tool for a managed domain or standalone server. It allows a user to connect to the domain controller or a standalone server and execute management operations available through the de-typed management model.

Details on how to use the CLI can be found in the Command Line Interface page.

5.2.3 Configuration Files

WildFly stores its configuration in centralized XML configuration files, one per server for standalone servers and, for managed domains, one per host with an additional domain wide policy controlled by the master host. These files are meant to be human-readable and human editable.

The XML configuration files act as a central, authoritative source of configuration. Any configuration changes made via the web interface or the CLI are persisted back to the XML configuration files. If a domain or standalone server is offline, the XML configuration files can be hand edited as well, and any changes will be picked up when the domain or standalone server is next started. However, users are encouraged to use the web interface or the CLI in preference to making offline edits to the configuration files. External changes made to the configuration files while processes are running will not be detected, and may be overwritten.

Standalone Server Configuration File

The XML configuration for a standalone server can be found in the `standalone/configuration` directory. The default configuration file is `standalone/configuration/standalone.xml`.

The `standalone/configuration` directory includes a number of other standard configuration files, e.g. `standalone-full.xml`, `standalone-ha.xml` and `standalone-full-ha.xml` each of which is similar to the default `standalone.xml` file but includes additional subsystems not present in the default configuration. If you prefer to use one of these files as your server configuration, you can specify it with the ~~*c*~~ or *-server-config* command line argument:

- `bin/standalone.sh -c=standalone-full.xml`
- `bin/standalone.sh --server-config=standalone-ha.xml`

Managed Domain Configuration Files

In a managed domain, the XML files are found in the `domain/configuration` directory. There are two types of configuration files – one per host, and then a single domain-wide file managed by the master host, aka the Domain Controller. (For more on the types of processes in a managed domain, see Operating modes.)

Host Specific Configuration – host.xml

When you start a managed domain process, a Host Controller instance is launched, and it parses its own configuration file to determine its own configuration, how it should integrate with the rest of the domain, any host-specific values for settings in the domain wide configuration (e.g. IP addresses) and what servers it should launch. This information is contained in the host-specific configuration file, the default version of which is `domain/configuration/host.xml`.

Each host will have its own variant `host.xml`, with settings appropriate for its role in the domain. WildFly ships with three standard variants:

host-master.xml	A configuration that specifies the Host Controller should become the master, aka the Domain Controller. No servers will be started by this Host Controller, which is a recommended setup for a production master.
host-slave.xml	A configuration that specifies the Host Controller should not become master and instead should register with a remote master and be controlled by it. This configuration launches servers, although a user will likely wish to modify how many servers are launched and what server groups they belong to.
host.xml	The default host configuration, tailored for an easy out of the box experience experimenting with a managed domain. This configuration specifies the Host Controller should become the master, aka the Domain Controller, but it also launches a couple of servers.

Which host-specific configuration should be used can be controlled via the _*--host-config*_ command line argument:

```
$ bin/domain.sh --host-config=host-master.xml
```

Domain Wide Configuration – domain.xml

Once a Host Controller has processed its host-specific configuration, it knows whether it is configured to act as the master Domain Controller. If it is, it must parse the domain wide configuration file, by default located at `domain/configuration/domain.xml`. This file contains the bulk of the settings that should be applied to the servers in the domain when they are launched – among other things, what subsystems they should run with what settings, what sockets should be used, and what deployments should be deployed.

Which domain-wide configuration should be used can be controlled via the _*--domain-config*_ command line argument:

```
$ bin/domain.sh --domain-config=domain-production.xml
```

That argument is only relevant for hosts configured to act as the master.

A slave Host Controller does not usually parse the domain wide configuration file. A slave gets the domain wide configuration from the remote master Domain Controller when it registers with it. A slave also will not persist changes to a `domain.xml` file if one is present on the filesystem. For that reason it is recommended that no `domain.xml` be kept on the filesystem of hosts that will only run as slaves.

A slave can be configured to keep a locally persisted copy of the domain wide configuration and then use it on boot (in case the master is not available.) See *--backup and --cached-dc* under Command line parameters.

5.3 Core management concepts

5.3.1 Operating modes

WildFly can be booted in two different modes. A *managed domain* allows you to run and manage a multi-server topology. Alternatively, you can run a *standalone server* instance.

Standalone Server

For many use cases, the centralized management capability available via a managed domain is not necessary. For these use cases, a WildFly instance can be run as a "standalone server". A standalone server instance is an independent process, much like an JBoss Application Server 3, 4, 5, or 6 instance is. Standalone instances can be launched via the `standalone.sh` or `standalone.bat` launch scripts.

If more than one standalone instance is launched and multi-server management is desired, it is the user's responsibility to coordinate management across the servers. For example, to deploy an application across all of the standalone servers, the user would need to individually deploy the application on each server.

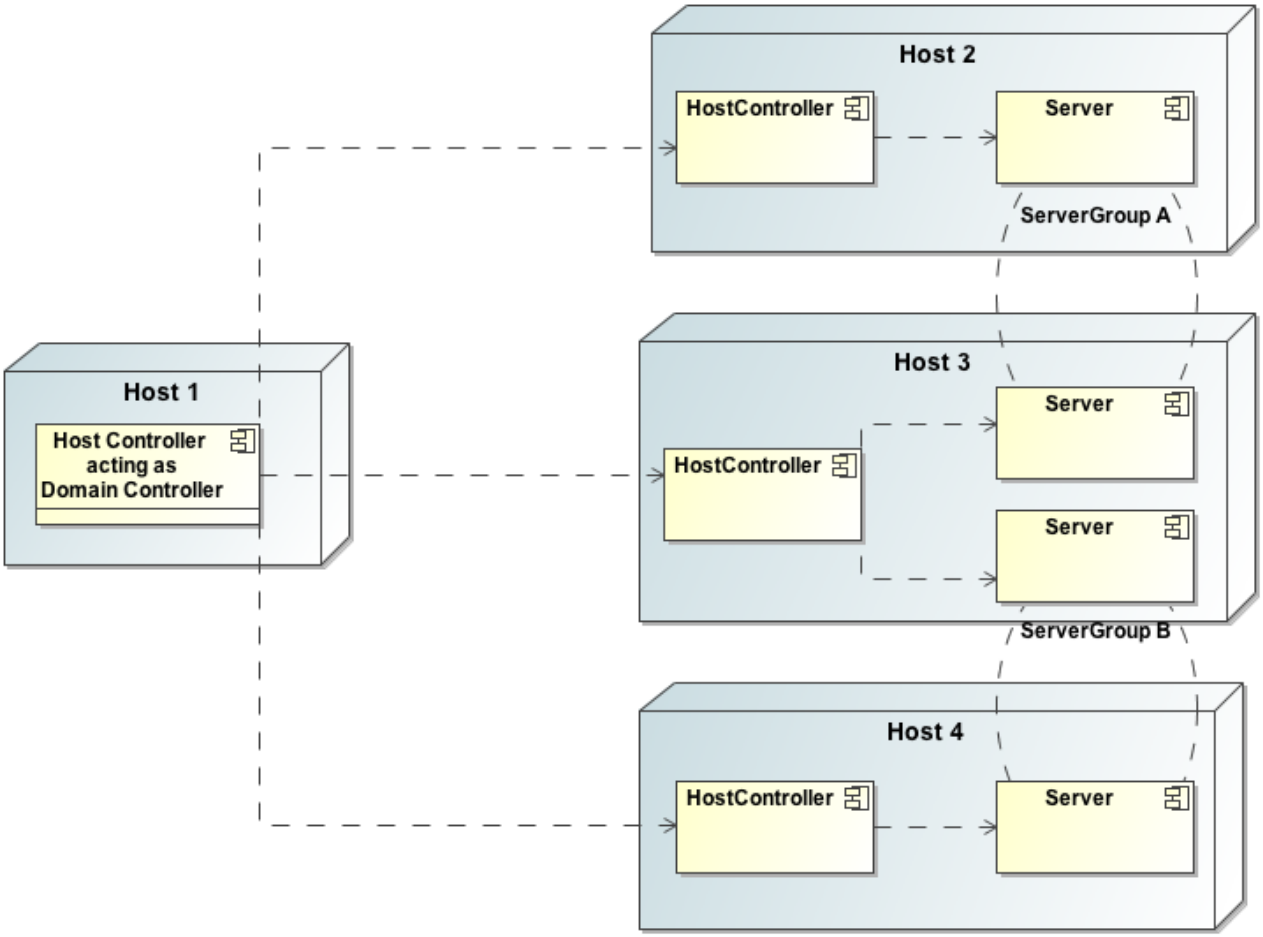
It is perfectly possible to launch multiple standalone server instances and have them form an HA cluster, just like it was possible with JBoss Application Server 3, 4, 5 and 6.

Managed Domain

One of the primary new features of WildFly is the ability to manage multiple WildFly instances from a single control point. A collection of such servers is referred to as the members of a "domain" with a single Domain Controller process acting as the central management control point. All of the WildFly instances in the domain share a common management policy, with the Domain Controller acting to ensure that each server is configured according to that policy. Domains can span multiple physical (or virtual) machines, with all WildFly instances on a given host under the control of a special Host Controller process. One Host Controller instance is configured to act as the central Domain Controller. The Host Controller on each host interacts with the Domain Controller to control the lifecycle of the application server instances running on its host and to assist the Domain Controller in managing them.

When you launch a WildFly managed domain on a host (via the `domain.sh` or `domain.bat` launch scripts) your intent is to launch a Host Controller and usually at least one WildFly instance. On one of the hosts the Host Controller should be configured to act as the Domain Controller. See Domain Setup for details.

The following is an example managed domain topology:



Host

Each "Host" box in the above diagram represents a physical or virtual host. A physical host can contain zero, one or more server instances.

Host Controller

When the domain.sh or domain.bat script is run on a host, a process known as a Host Controller is launched. The Host Controller is solely concerned with server management; it does not itself handle application server workloads. The Host Controller is responsible for starting and stopping the individual application server processes that run on its host, and interacts with the Domain Controller to help manage them.

Each Host Controller by default reads its configuration from the `domain/configuration/host.xml` file located in the unzipped WildFly installation on its host's filesystem. The `host.xml` file contains configuration information that is specific to the particular host. Primarily:

- the listing of the names of the actual WildFly instances that are meant to run off of this installation.
- configuration of how the Host Controller is to contact the Domain Controller to register itself and access the domain configuration. This may either be configuration of how to find and contact a remote Domain Controller, or a configuration telling the Host Controller to itself act as the Domain Controller.
- configuration of items that are specific to the local physical installation. For example, named interface definitions declared in `domain.xml` (see below) can be mapped to an actual machine-specific IP address in `host.xml`. Abstract path names in `domain.xml` can be mapped to actual filesystem paths in `host.xml`.

Domain Controller

One Host Controller instance is configured to act as the central management point for the entire domain, i.e. to be the Domain Controller. The primary responsibility of the Domain Controller is to maintain the domain's central management policy, to ensure all Host Controllers are aware of its current contents, and to assist the Host Controllers in ensuring any running application server instances are configured in accordance with this policy. This central management policy is stored by default in the `domain/configuration/domain.xml` file in the unzipped WildFly installation on Domain Controller's host's filesystem.

A `domain.xml` file must be located in the `domain/configuration` directory of an installation that's meant to run the Domain Controller. It does not need to be present in installations that are not meant to run a Domain Controller; i.e. those whose Host Controller is configured to contact a remote Domain Controller. The presence of a `domain.xml` file on such a server does no harm.

The `domain.xml` file includes, among other things, the configuration of the various "profiles" that WildFly instances in the domain can be configured to run. A profile configuration includes the detailed configuration of the various subsystems that comprise that profile (e.g. an embedded JBoss Web instance is a subsystem; a JBoss TS transaction manager is a subsystem, etc). The domain configuration also includes the definition of groups of sockets that those subsystems may open. The domain configuration also includes the definition of "server groups":

Server Group

A server group is set of server instances that will be managed and configured as one. In a managed domain each application server instance is a member of a server group. (Even if the group only has a single server, the server is still a member of a group.) It is the responsibility of the Domain Controller and the Host Controllers to ensure that all servers in a server group have a consistent configuration. They should all be configured with the same profile and they should have the same deployment content deployed.

The domain can have multiple server groups. The above diagram shows two server groups, "ServerGroupA" and "ServerGroupB". Different server groups can be configured with different profiles and deployments; for example in a domain with different tiers of servers providing different services. Different server groups can also run the same profile and have the same deployments; for example to support rolling application upgrade scenarios where a complete service outage is avoided by first upgrading the application on one server group and then upgrading a second server group.

An example server group definition is as follows:

```
<server-group name="main-server-group" profile="default">
    <socket-binding-group ref="standard-sockets"/>
    <deployments>
        <deployment name="foo.war_v1" runtime-name="foo.war" />
        <deployment name="bar.ear" runtime-name="bar.ear" />
    </deployments>
</server-group>
```

A server-group configuration includes the following required attributes:

- name -- the name of the server group
- profile -- the name of the profile the servers in the group should run

In addition, the following optional elements are available:

- socket-binding-group -- specifies the name of the default socket binding group to use on servers in the group. Can be overridden on a per-server basis in `host.xml`. If not provided in the `server-group` element, it must be provided for each server in `host.xml`.
- deployments -- the deployment content that should be deployed on the servers in the group.
- deployment-overlays -- the overlays and their associated deployments.
- system-properties -- system properties that should be set on all servers in the group
- jvm -- default jvm settings for all servers in the group. The Host Controller will merge these settings with any provided in `host.xml` to derive the settings to use to launch the server's JVM. See JVM settings for further details.

Server

Each "Server" in the above diagram represents an actual application server instance. The server runs in a separate JVM process from the Host Controller. The Host Controller is responsible for launching that process. (In a managed domain the end user cannot directly launch a server process from the command line.)

The Host Controller synthesizes the server's configuration by combining elements from the domain wide configuration (from `domain.xml`) and the host-specific configuration (from `host.xml`).

Deciding between running standalone servers or a managed domain

Which use cases are appropriate for managed domain and which are appropriate for standalone servers? A managed domain is all about coordinated multi-server management -- with it WildFly provides a central point through which users can manage multiple servers, with rich capabilities to keep those servers' configurations consistent and the ability to roll out configuration changes (including deployments) to the servers in a coordinated fashion.

It's important to understand that the choice between a managed domain and standalone servers is all about how your servers are managed, not what capabilities they have to service end user requests. This distinction is particularly important when it comes to high availability clusters. It's important to understand that HA functionality is orthogonal to running standalone servers or a managed domain. That is, a group of standalone servers can be configured to form an HA cluster. The domain and standalone modes determine how the servers are managed, not what capabilities they provide.

So, given all that:

- A single server installation gains nothing from running in a managed domain, so running a standalone server is a better choice.
- For multi-server production environments, the choice of running a managed domain versus standalone servers comes down to whether the user wants to use the centralized management capabilities a managed domain provides. Some enterprises have developed their own sophisticated multi-server management capabilities and are comfortable coordinating changes across a number of independent WildFly instances. For these enterprises, a multi-server architecture comprised of individual standalone servers is a good option.
- Running a standalone server is better suited for most development scenarios. Any individual server configuration that can be achieved in a managed domain can also be achieved in a standalone server, so even if the application being developed will eventually run in production on a managed domain installation, much (probably most) development can be done using a standalone server.
- Running a managed domain mode can be helpful in some advanced development scenarios; i.e. those involving interaction between multiple WildFly instances. Developers may find that setting up various servers as members of a domain is an efficient way to launch a multi-server cluster.

5.3.2 General configuration concepts

For both a managed domain or a standalone server, a number of common configuration concepts apply:

Extensions

An extension is a module that extends the core capabilities of the server. The WildFly core is very simple and lightweight; most of the capabilities people associate with an application server are provided via extensions. An extension is packaged as a module in the `modules` folder. The user indicates that they want a particular extension to be available by including an `<extension/>` element naming its module in the `domain.xml` or `standalone.xml` file.

```
<extensions>
    [...]
    <extension module="org.jboss.as.transactions"/>
    <extension module="org.jboss.as.webservices" />
    <extension module="org.jboss.as.weld" />
    [...]
    <extension module="org.wildfly.extension.undertow"/>
</extensions>
```

Profiles and Subsystems

The most significant part of the configuration in `domain.xml` and `standalone.xml` is the configuration of one (in `standalone.xml`) or more (in `domain.xml`) "profiles". A profile is a named set of subsystem configurations. A subsystem is an added set of capabilities added to the core server by an extension (see "Extensions" above). A subsystem provides servlet handling capabilities; a subsystem provides an EJB container; a subsystem provides JTA, etc. A profile is a named list of subsystems, along with the details of each subsystem's configuration. A profile with a large number of subsystems results in a server with a large set of capabilities. A profile with a small, focused set of subsystems will have fewer capabilities but a smaller footprint.

The content of an individual profile configuration looks largely the same in `domain.xml` and `standalone.xml`. The only difference is `standalone.xml` is only allowed to have a single profile element (the profile the server will run), while `domain.xml` can have many profiles, each of which can be mapped to one or more groups of servers.

The contents of individual subsystem configurations look exactly the same between `domain.xml` and `standalone.xml`.

Paths

A logical name for a filesystem path. The `domain.xml`, `host.xml` and `standalone.xml` configurations all include a section where paths can be declared. Other sections of the configuration can then reference those paths by their logical name, rather than having to include the full details of the path (which may vary on different machines). For example, the logging subsystem configuration includes a reference to the "`jboss.server.log.dir`" path that points to the server's "`log`" directory.

```
<file relative-to="jboss.server.log.dir" path="server.log"/>
```

WildFly automatically provides a number of standard paths without any need for the user to configure them in a configuration file:

- `jboss.home.dir` - the root directory of the WildFly distribution
- `user.home` - user's home directory
- `user.dir` - user's current working directory
- `java.home` - java installation directory
- `jboss.server.base.dir` - root directory for an individual server instance
- `jboss.server.config.dir` - directory the server will use for configuration file storage
- `jboss.server.data.dir` - directory the server will use for persistent data file storage
- `jboss.server.log.dir` - directory the server will use for log file storage
- `jboss.server.temp.dir` - directory the server will use for temporary file storage
- `jboss.controller.temp.dir` - directory the server will use for temporary file storage
- `jboss.domain.servers.dir` - directory under which a host controller will create the working area for individual server instances (managed domain mode only)

Users can add their own paths or override all except the first 5 of the above by adding a `<path/>` element to their configuration file.

```
<path name="example" path="example" relative-to="jboss.server.data.dir"/>
```

The attributes are:

- `name` -- the name of the path.
- `path` -- the actual filesystem path. Treated as an absolute path, unless the 'relative-to' attribute is specified, in which case the value is treated as relative to that path.
- `relative-to` -- (optional) the name of another previously named path, or of one of the standard paths provided by the system.

A `<path/>` element in a `domain.xml` need not include anything more than the `name` attribute; i.e. it need not include any information indicating what the actual filesystem path is:

```
<path name="x"/>
```

Such a configuration simply says, "There is a path named 'x' that other parts of the `domain.xml` configuration can reference. The actual filesystem location pointed to by 'x' is host-specific and will be specified in each machine's `host.xml` file." If this approach is used, there must be a path element in each machine's `host.xml` that specifies what the actual filesystem path is:

```
<path name="x" path="/var/x" />
```

A <path/> element in a standalone.xml must include the specification of the actual filesystem path.

Interfaces

A logical name for a network interface/IP address/host name to which sockets can be bound. The domain.xml, host.xml and standalone.xml configurations all include a section where interfaces can be declared. Other sections of the configuration can then reference those interfaces by their logical name, rather than having to include the full details of the interface (which may vary on different machines). An interface configuration includes the logical name of the interface as well as information specifying the criteria to use for resolving the actual physical address to use. See Interfaces and ports for further details.

An <interface/> element in a domain.xml need not include anything more than the name attribute; i.e. it need not include any information indicating what the actual IP address associated with the name is:

```
<interface name="internal"/>
```

Such a configuration simply says, "There is an interface named 'internal' that other parts of the domain.xml configuration can reference. The actual IP address pointed to by 'internal' is host-specific and will be specified in each machine's host.xml file." If this approach is used, there must be an interface element in each machine's host.xml that specifies the criteria for determining the IP address:

```
<interface name="internal">
   <nic name="eth1"/>
</interface>
```

An <interface/> element in a standalone.xml must include the criteria for determining the IP address.

Socket Bindings and Socket Binding Groups

A socket binding is a named configuration for a socket.

The domain.xml and standalone.xml configurations both include a section where named socket configurations can be declared. Other sections of the configuration can then reference those sockets by their logical name, rather than having to include the full details of the socket configuration (which may vary on different machines). See Interfaces and ports for full details.

System Properties

System property values can be set in a number of places in `domain.xml`, `host.xml` and `standalone.xml`. The values in `standalone.xml` are set as part of the server boot process. Values in `domain.xml` and `host.xml` are applied to servers when they are launched.

When a system property is configured in `domain.xml` or `host.xml`, the servers it ends up being applied to depends on where it is set. Setting a system property in a child element directly under the `domain.xml` root results in the property being set on all servers. Setting it in a `<system-property/>` element inside a `<server-group/>` element in domain.xml results in the property being set on all servers in the group. Setting it in a child element directly under the `host.xml` root results in the property being set on all servers controlled by that host's Host Controller. Finally, setting it in a `<system-property/>` element inside a `<server/>` element in `host.xml` result in the property being set on that server. The same property can be configured in multiple locations, with a value in a `<server/>` element taking precedence over a value specified directly under the `host.xml` root element, the value in a `host.xml` taking precedence over anything from `domain.xml`, and a value in a `<server-group/>` element taking precedence over a value specified directly under the `domain.xml` root element.

5.3.3 Management resources

When WildFly parses your configuration files at boot, or when you use one of the AS's Management Clients you are adding, removing or modifying *management resources* in the AS's internal management model. A WildFly management resource has the following characteristics:

Address

All WildFly management resources are organized in a tree. The path to the node in the tree for a particular resource is its *address*. Each segment in a resource's address is a key/value pair:

- The key is the resource's *type*, in the context of its parent. So, for example, the root resource for a standalone server has children of type `subsystem`, `interface`, `socket-binding`, etc. The resource for the subsystem that provides the AS's webserver capability has children of type `connector` and `virtual-server`. The resource for the subsystem that provides the AS's messaging server capability has, among others, children of type `jms-queue` and `jms-topic`.
- The value is the name of a particular resource of the given type, e.g `web` or `messaging` for subsystems or `http` or `https` for web subsystem connectors.

The full address for a resource is the ordered list of key/value pairs that lead from the root of the tree to the resource. Typical notation is to separate the elements in the address with a '/' and to separate the key and the value with an '=':

- `/subsystem=undertow/server=default-server/http-listener=default`
- `/subsystem=messaging/jms-queue=testQueue`
- `/interface=public`

When using the HTTP API, a '/' is used to separate the key and the value instead of an '=':

- `http://localhost:9990/management/subsystem/undertow/server/default-server/h`
- `http://localhost:9990/management/subsystem/messaging/jms-queue/testQueue`
- `http://localhost:9990/management/interface/public`

Operations

Querying or modifying the state of a resource is done via an operation. An operation has the following characteristics:

- A string name
- Zero or more named parameters. Each parameter has a string name, and a value of type `org.jboss.dmr.ModelNode` (or, when invoked via the CLI, the text representation of a `ModelNode`; when invoked via the HTTP API, the JSON representation of a `ModelNode`.) Parameters may be optional.
- A return value, which will be of type `org.jboss.dmr.ModelNode` (or, when invoked via the CLI, the text representation of a `ModelNode`; when invoked via the HTTP API, the JSON representation of a `ModelNode`.)

Every resource except the root resource will have an `add` operation and should have a `remove` operation ("should" because in WildFly 8 many do not). The parameters for the `add` operation vary depending on the resource. The `remove` operation has no parameters.

There are also a number of "global" operations that apply to all resources. See Global operations for full details.

The operations a resource supports can themselves be determined by invoking an operation: the `read-operation-names` operation. Once the name of an operation is known, details about its parameters and return value can be determined by invoking the `read-operation-description` operation. For example, to learn the names of the operations exposed by the root resource for a standalone server, and then learn the full details of one of them, via the CLI one would:

```
[standalone@localhost:9990 /] :read-operation-names
{
    "outcome" => "success",
    "result" => [
        "add-namespace",
        "add-schema-location",
        "delete-snapshot",
        "full-replace-deployment",
        "list-snapshots",
        "read-attribute",
        "read-children-names",
        "read-children-resources",
        "read-children-types",
        "read-config-as-xml",
        "read-operation-description",
        "read-operation-names",
        "read-resource",
        "read-resource-description",
        "reload",
        "remove-namespace",
        "remove-schema-location",
        "replace-deployment",
        "shutdown",
        "take-snapshot",
        "upload-deployment-bytes",
        "upload-deployment-stream",
        "upload-deployment-url",
        "validate-address",
        "write-attribute"
    ]
}
[standalone@localhost:9990 /] :read-operation-description(name=upload-deployment-url)
{
    "outcome" => "success",
    "result" => {
        "operation-name" => "upload-deployment-url",
        "description" => "Indicates that the deployment content available at the included URL
should be added to the deployment content repository. Note that this operation does not indicate
the content should be deployed into the runtime.",
        "request-properties" => {"url" => {
            "type" => STRING,
            "description" => "The URL at which the deployment content is available for upload to
the domain's or standalone server's deployment content repository.. Note that the URL must be
accessible from the target of the operation (i.e. the Domain Controller or standalone server).",
            "required" => true,
            "min-length" => 1,
            "nillable" => false
        }},
        "reply-properties" => {
            "type" => BYTES,
            "description" => "The hash of managed deployment content that has been uploaded to
the domain's or standalone server's deployment content repository.",
            "min-length" => 20,
            "max-length" => 20,
            "nillable" => false
        }
    }
}
```

See Descriptions below for more on how to learn about the operations a resource exposes.

Attributes

Management resources expose information about their state as attributes. Attributes have string name, and a value of type `org.jboss.dmr.ModelNode` (or: for the CLI, the text representation of a `ModelNode`; for HTTP API, the JSON representation of a `ModelNode`.)

Attributes can either be read-only or read-write. Reading and writing attribute values is done via the global `read-attribute` and `write-attribute` operations.

The `read-attribute` operation takes a single parameter "name" whose value is a the name of the attribute. For example, to read the "port" attribute of a socket-binding resource via the CLI:

```
[standalone@localhost:9990 /]
/socket-binding-group=standard-sockets/socket-binding=https:read-attribute(name=port)
{
    "outcome" => "success",
    "result" => 8443
}
```

If an attribute is writable, the `write-attribute` operation is used to mutate its state. The operation takes two parameters:

- `name` – the name of the attribute
- `value` – the value of the attribute

For example, to read the "port" attribute of a socket-binding resource via the CLI:

```
[standalone@localhost:9990 /]
/socket-binding-group=standard-sockets/socket-binding=https:write-attribute(name=port,value=8444)
=> "success"}
```

Attributes can have one of two possible *storage types*:

- **CONFIGURATION** – means the value of the attribute is stored in the persistent configuration; i.e. in the `domain.xml`, `host.xml` or `standalone.xml` file from which the resource's configuration was read.
- **RUNTIME** – the attribute value is only available from a running server; the value is not stored in the persistent configuration. A metric (e.g. number of requests serviced) is a typical example of a RUNTIME attribute.

The values of all of the attributes a resource exposes can be obtained via the `read-resource` operation, with the "include-runtime" parameter set to "true". For example, from the CLI:

```
[standalone@localhost:9990 /]
/subsystem=undertow/server=default-server/http-listener=default:read-resource(include-runtime=true
"outcome" => "success",
    "result" => {
        "allow-encoded-slash" => false,
        "allow-equals-in-cookie-value" => false,
        "always-set-keep-alive" => true,
        "buffer-pipelined-data" => true,
        "buffer-pool" => "default",
        "bytes-received" => 0L,
        "bytes-sent" => 0L,
        "certificate-forwarding" => false,
        "decode-url" => true,
        "disallowed-methods" => ["TRACE"],
        "enable-http2" => false,
        "enabled" => true,
        "error-count" => 0L,
        "max-buffered-request-size" => 16384,
        "max-connections" => undefined,
        "max-cookies" => 200,
        "max-header-size" => 1048576,
        "max-headers" => 200,
        "max-parameters" => 1000,
        "max-post-size" => 10485760L,
        "max-processing-time" => 0L,
        "no-request-timeout" => undefined,
        "processing-time" => 0L,
        "proxy-address-forwarding" => false,
        "read-timeout" => undefined,
        "receive-buffer" => undefined,
        "record-request-start-time" => false,
        "redirect-socket" => "https",
        "request-count" => 0L,
        "request-parse-timeout" => undefined,
        "resolve-peer-address" => false,
        "send-buffer" => undefined,
        "socket-binding" => "http",
        "tcp-backlog" => undefined,
        "tcp-keep-alive" => undefined,
        "url-charset" => "UTF-8",
        "worker" => "default",
        "write-timeout" => undefined
    }
}
```

Omit the "include-runtime" parameter (or set it to "false") to limit output to those attributes whose values are stored in the persistent configuration:

```
[standalone@localhost:9990 /]
/subsystem=undertow/server=default-server/http-listener=default:read-resource(include-runtime=fals
"outcome" => "success",
    "result" => {
        "allow-encoded-slash" => false,
        "allow-equals-in-cookie-value" => false,
        "always-set-keep-alive" => true,
        "buffer-pipelined-data" => true,
        "buffer-pool" => "default",
        "certificate-forwarding" => false,
        "decode-url" => true,
        "disallowed-methods" => ["TRACE"],
        "enable-http2" => false,
        "enabled" => true,
        "max-buffered-request-size" => 16384,
        "max-connections" => undefined,
        "max-cookies" => 200,
        "max-header-size" => 1048576,
        "max-headers" => 200,
        "max-parameters" => 1000,
        "max-post-size" => 10485760L,
        "no-request-timeout" => undefined,
        "proxy-address-forwarding" => false,
        "read-timeout" => undefined,
        "receive-buffer" => undefined,
        "record-request-start-time" => false,
        "redirect-socket" => "https",
        "request-parse-timeout" => undefined,
        "resolve-peer-address" => false,
        "send-buffer" => undefined,
        "socket-binding" => "http",
        "tcp-backlog" => undefined,
        "tcp-keep-alive" => undefined,
        "url-charset" => "UTF-8",
        "worker" => "default",
        "write-timeout" => undefined
    }
}
```

See Descriptions below for how to learn more about the attributes a particular resource exposes.

Children

Management resources may support child resources. The *types* of children a resource supports (e.g. `connector` for the web subsystem resource) can be obtained by querying the resource's description (see Descriptions below) or by invoking the `read-children-types` operation. Once you know the legal child types, you can query the names of all children of a given type by using the global `read-children-types` operation. The operation takes a single parameter "child-type" whose value is the type. For example, a resource representing a socket binding group has children. To find the type of those children and the names of resources of that type via the CLI one could:

```
[standalone@localhost:9990 /] /socket-binding-group=standard-sockets:read-children-types
{
    "outcome" => "success",
    "result" => ["socket-binding"]
}
[standalone@localhost:9990 /]
/socket-binding-group=standard-sockets:read-children-names(child-type=socket-binding)
{
    "outcome" => "success",
    "result" => [
        "http",
        "https",
        "jmx-connector-registry",
        "jmx-connector-server",
        "jndi",
        "osgi-http",
        "remoting",
        "txn-recovery-environment",
        "txn-status-manager"
    ]
}
```

Descriptions

All resources expose metadata that describes their attributes, operations and child types. This metadata is itself obtained by invoking one or more of the global operations each resource supports. We showed examples of the `read-operation-names`, `read-operation-description`, `read-children-types` and `read-children-names` operations above.

The `read-resource-description` operation can be used to find the details of the attributes and child types associated with a resource. For example, using the CLI:

```
[standalone@localhost:9990 /] /socket-binding-group=standard-sockets:read-resource-description
{
    "outcome" => "success",
    "result" => {
        "description" => "Contains a list of socket configurations.",
        "head-comment-allowed" => true,
        "tail-comment-allowed" => false,
        "attributes" => {
            "name" => {
                "type" => STRING,
                "description" => "The name of the socket binding group.",
                "required" => true,
                "head-comment-allowed" => false,
                "tail-comment-allowed" => false,
                "access-type" => "read-only",
                "storage" => "configuration"
            },
            "default-interface" => {
                "type" => STRING,
                "description" => "Name of an interface that should be used as the interface for
any sockets that do not explicitly declare one.",
                "required" => true,
                "head-comment-allowed" => false,
                "tail-comment-allowed" => false,
                "access-type" => "read-write",
                "storage" => "configuration"
            },
            "port-offset" => {
                "type" => INT,
                "description" => "Increment to apply to the base port values defined in the
socket bindings to derive the runtime values to use on this server.",
                "required" => false,
                "head-comment-allowed" => true,
                "tail-comment-allowed" => false,
                "access-type" => "read-write",
                "storage" => "configuration"
            }
        },
        "operations" => {},
        "children" => {"socket-binding" => {
            "description" => "The individual socket configurtions.",
            "min-occurs" => 0,
            "model-description" => undefined
        }}
    }
}
```

Note the `"operations" => }}` in the output above. If the command had included the `{{operations` parameter (i.e. `/socket-binding-group=standard-sockets:read-resource-description(operations=true`) the output would have included the description of each operation supported by the resource.

See the Global operations section for details on other parameters supported by the `read-resource-description` operation and all the other globally available operations.

Comparison to JMX MBeans

WildFly management resources are conceptually quite similar to Open MBeans. They have the following primary differences:

- WildFly management resources are organized in a tree structure. The order of the key value pairs in a resource's address is significant, as it defines the resource's position in the tree. The order of the key properties in a JMX `ObjectName` is not significant.
- In an Open MBean attribute values, operation parameter values and operation return values must either be one of the simple JDK types (String, Boolean, Integer, etc) or implement either the `javax.management.openmbean.CompositeData` interface or the `javax.management.openmbean.TabularData` interface. WildFly management resource attribute values, operation parameter values and operation return values are all of type `org.jboss.dmr.ModelNode`.

Basic structure of the management resource trees

As noted above, management resources are organized in a tree structure. The structure of the tree depends on whether you are running a standalone server or a managed domain.

Standalone server

The structure of the managed resource tree is quite close to the structure of the `standalone.xml` configuration file.

- The root resource
  - `extension` – extensions installed in the server
  - `path` – paths available on the server
  - `system-property` – system properties set as part of the configuration (i.e. not on the command line)
  - `core-service=management` – the server's core management services
  - `core-service=service-container` – resource for the JBoss MSC `ServiceContainer` that's at the heart of the AS
  - `subsystem` – the subsystems installed on the server. The bulk of the management model will be children of type `subsystem`
  - `interface` – interface configurations
  - `socket-binding-group` – the central resource for the server's socket bindings
    - `socket-binding` – individual socket binding configurations
  - `deployment` – available deployments on the server

Managed domain

In a managed domain, the structure of the managed resource tree spans the entire domain, covering both the domain wide configuration (e.g. what's in `domain.xml`, the host specific configuration for each host (e.g. what's in `host.xml`, and the resources exposed by each running application server. The Host Controller processes in a managed domain provide access to all or part of the overall resource tree. How much is available depends on whether the management client is interacting with the Host Controller that is acting as the master Domain Controller. If the Host Controller is the master Domain Controller, then the section of the tree for each host is available. If the Host Controller is a slave to a remote Domain Controller, then only the portion of the tree associated with that host is available.

- The root resource for the entire domain. The persistent configuration associated with this resource and its children, except for those of type `host`, is persisted in the `domain.xml` file on the Domain Controller.

- `extension` – extensions available in the domain
- `path` – paths available on across the domain
- `system-property` – system properties set as part of the configuration (i.e. not on the command line) and available across the domain
- `profile` – sets of subsystem configurations that can be assigned to server groups
    - `subsystem` – configuration of subsystems that are part of the profile
- `interface` – interface configurations
- `socket-binding-group` – sets of socket bindings configurations that can be applied to server groups
    - `socket-binding` – individual socket binding configurations
- `deployment` – deployments available for assignment to server groups
- deployment-overlay -- deployment-overlays content available to overlay deployments in server groups
- `server-group` – server group configurations
- host – the individual Host Controllers. Each child of this type represents the root resource for a particular host. The persistent configuration associated with one of these resources or its children is persisted in the host's `host.xml` file.
    - `path` – paths available on each server on the host
    - `system-property` – system properties to set on each server on the host
    - `core-service=management` – the Host Controller's core management services
    - `interface` – interface configurations that apply to the Host Controller or servers on the host
    - `jvm` – JVM configurations that can be applied when launching servers
    - `server-config` – configuration describing how the Host Controller should launch a server; what server group configuration to use, and any server-specific overrides of items specified in other resources
    - `server` – the root resource for a running server. Resources from here and below are not directly persisted; the domain-wide and host level resources contain the persistent configuration that drives a server
        - `extension` – extensions installed in the server
        - `path` – paths available on the server
        - `system-property` – system properties set as part of the configuration (i.e. not on the command line)
        - `core-service=management` – the server's core management services
        - `core-service=service-container` – resource for the JBoss MSC `ServiceContainer` that's at the heart of the AS
        - `subsystem` – the subsystems installed on the server. The bulk of the management model will be children of type `subsystem`
        - `interface` – interface configurations
        - `socket-binding-group` – the central resource for the server's socket bindings
            - `socket-binding` – individual socket binding configurations
        - `deployment` – available deployments on the server
        - deployment-overlay -- available overlays on the server

5.4 Configuring interfaces and ports

5.4.1 Interface declarations

WildFly uses named interface references throughout the configuration. A network interface is declared by specifying a logical name and a selection criteria for the physical interface:

```
[standalone@localhost:9990 /] :read-children-names(child-type=interface)
{
   "outcome" => "success",
   "result" => [
       "management",
       "public"
   ]
}
```

This means the server in question declares two interfaces: One is referred to as "*management*"; the other one "*public*". The "*management*" interface is used for all components and services that are required by the management layer (i.e. the HTTP Management Endpoint). The "*public*" interface binding is used for any application related network communication (i.e. Web, Messaging, etc). There is nothing special about these names; interfaces can be declared with any name. Other sections of the configuration can then reference those interfaces by their logical name, rather than having to include the full details of the interface (which, on servers in a management domain, may vary on different machines).

The `domain.xml`, `host.xml` and `standalone.xml` configuration files all include a section where interfaces can be declared. If we take a look at the XML declaration it reveals the selection criteria. The criteria is one of two types: either a single element indicating that the interface should be bound to a wildcard address, or a set of one or more characteristics that an interface or address must have in order to be a valid match. The selection criteria in this example are specific IP addresses for each interface:

```
<interfaces>
  <interface name="management">
   <inet-address value="127.0.0.1"/>
  </interface>
  <interface name="public">
   <inet-address value="127.0.0.1"/>
  </interface>
</interfaces>
```

Some other examples:

```
<interface name="global">
   <!-- Use the wildcard address -->
   <any-address/>
</interface>

<interface name="external">
   <nic name="eth0"/>
</interface>

<interface name="default">
   <!-- Match any interface/address on the right subnet if it's
        up, supports multicast and isn't point-to-point -->
   <subnet-match value="192.168.0.0/16"/>
   <up/>
   <multicast/>
   <not>
      <point-to-point/>
   </not>
</interface>
```

The -b command line argument

WildFly supports using the `-b` command line argument to specify the address to assign to interfaces. See Controlling the Bind Address with -b for further details.

5.4.2 Socket Binding Groups

The socket configuration in WildFly works similarly to the interfaces declarations. Sockets are declared using a logical name, by which they will be referenced throughout the configuration. Socket declarations are grouped under a certain name. This allows you to easily reference a particular socket binding group when configuring server groups in a managed domain. Socket binding groups reference an interface by its logical name:

```
<socket-binding-group name="standard-sockets" default-interface="public">
  <socket-binding name="management-http" interface="management"
port="${jboss.management.http.port:9990}"/>
  <socket-binding name="management-https" interface="management"
port="${jboss.management.https.port:9993}"/>
  <socket-binding name="ajp" port="${jboss.ajp.port:8009}"/>
  <socket-binding name="http" port="${jboss.http.port:8080}"/>
  <socket-binding name="https" port="${jboss.https.port:8443}"/>
  <socket-binding name="txn-recovery-environment" port="4712"/>
  <socket-binding name="txn-status-manager" port="4713"/>
</socket-binding-group>
```

A socket binding includes the following information:

- name -- logical name of the socket configuration that should be used elsewhere in the configuration
- port -- base port to which a socket based on this configuration should be bound. (Note that servers can be configured to override this base value by applying an increment or decrement to all port values.)
- interface (optional) -- logical name (see "Interfaces declarations" above) of the interface to which a socket based on this configuration should be bound. If not defined, the value of the "default-interface" attribute from the enclosing socket binding group will be used.
- multicast-address (optional) -- if the socket will be used for multicast, the multicast address to use
- multicast-port (optional) -- if the socket will be used for multicast, the multicast port to use
- fixed-port (optional, defaults to false) -- if true, declares that the value of port should always be used for the socket and should not be overridden by applying an increment or decrement

5.4.3 IPv4 versus IPv6

WildFly supports the use of both IPv4 and IPv6 addresses. By default, WildFly is configured for use in an IPv4 network and so if you are running in an IPv4 network, no changes are required. If you need to run in an IPv6 network, the changes required are minimal and involve changing the JVM stack and address preferences, and adjusting any interface IP address values specified in the configuration (standalone.xml or domain.xml).

Stack and address preference

The system properties java.net.preferIPv4Stack and java.net.preferIPv6Addresses are used to configure the JVM for use with IPv4 or IPv6 addresses. With WildFly, in order to run using IPv4 addresses, you need to specify java.net.preferIPv4Stack=true; in order to run with IPv6 addresses, you need to specify java.net.preferIPv4Stack=false (the JVM default) and java.net.preferIPv6Addresses=true. The latter ensures that any hostname to IP address conversions always return IPv6 address variants.

These system properties are conveniently set by the JAVA_OPTS environment variable, defined in the standalone.conf (or domain.conf) file. For example, to change the IP stack preference from its default of IPv4 to IPv6, edit the standalone.conf (or domain.conf) file and change its default IPv4 setting:

```
if [ "x$JAVA_OPTS" = "x" ]; then
   JAVA_OPTS=" ... -Djava.net.preferIPv4Stack=true ..."
...
```

to an IPv6 suitable setting:

```
if [ "x$JAVA_OPTS" = "x" ]; then
   JAVA_OPTS=" ... -Djava.net.preferIPv4Stack=false -Djava.net.preferIPv6Addresses=true ..."
...
```

IP address literals

To change the IP address literals referenced in standalone.xml (or domain.xml), first visit the interface declarations and ensure that valid IPv6 addresses are being used as interface values. For example, to change the default configuration in which the loopback interface is used as the primary interface, change from the IPv4 loopback address:

```
<interfaces>
  <interface name="management">
    <inet-address value="${jboss.bind.address.management:127.0.0.1}"/>
  </interface>
  <interface name="public">
    <inet-address value="${jboss.bind.address:127.0.0.1}"/>
  </interface>
</interfaces>
```

to the IPv6 loopback address:

```
<interfaces>
  <interface name="management">
    <inet-address value="${jboss.bind.address.management:[::1]}"/>
  </interface>
  <interface name="public">
    <inet-address value="${jboss.bind.address:[::1]}"/>
  </interface>
</interfaces>
```

Note that when embedding IPv6 address literals in the substitution expression, square brackets surrounding the IP address literal are used to avoid ambiguity. This follows the convention for the use of IPv6 literals in URLs.

Over and above making such changes for the interface definitions, you should also check the rest of your configuration file and adjust IP address literals from IPv4 to IPv6 as required.

5.5 Administrative security

5.5.1 Security realms

Within WildFly we make use of security realms to secure access to the management interfaces, these same realms are used to secure inbound access as exposed by JBoss Remoting such as remote JNDI and EJB access, the realms are also used to define an identity for the server - this identity can be used for both inbound connections to the server and outbound connections being established by the server.

General Structure

The general structure of a management realm definition is: -

```
<security-realm name="ManagementRealm">
   <plug-ins></plug-ins>
   <server-identities></server-identities>
   <authentication></authentication>
   <authorization></authorization>
</security-realm>
```

- `plug-ins` - This is an optional element that is used to define modules what will be searched for security realm PlugInProviders to extend the capabilities of the security realms.
- `server-identities` - An optional element to define the identity of the server as visible to the outside world, this applies to both inbound connection to a resource secured by the realm and to outbound connections also associated with the realm.

One example is the SSL identity of the server, for inbound connections this will control the identity of the server as the SSL connection is established, for outbound connections this same identity can be used where CLIENT-CERT style authentication is being performed.

A second example is where the server is establishing an outbound connection that requires username / password authentication - this element can be used to define that password.

- `authentication` - This is probably the most important element that will be used within a security realm definition and mostly applies to inbound connections to the server, this element defines which backing stores will be used to provide the verification of the inbound connection.

This element is optional as there are some scenarios where it will not be required such as if a realm is being defined for an outbound connection using a username and password.

- `authorization` - This is the final optional element and is used to define how roles are loaded for an authenticated identity. At the moment this is more applicable for realms used for access to EE deployments such as web applications or EJBs but this will also become relevant as we add role based authorization checks to the management model.

Using a Realm

After a realm has been defined it needs to be associated with an inbound or outbound connection for it to be used, the following are some examples where these associations are used within the WildFly 8 configuration.

Inbound Connections

Management Interfaces

Either within the `standalone.xml` or `host.xml` configurations the security realms can be associated with the management interface as follows:

```
<http-interface security-realm="ManagementRealm">...</http-interface>
```

If the `security-realm` attribute is omitted or removed from the interface definition it means that access to that interface will be unsecured.

By default we do bind these interfaces to the loopback address so that the interfaces are not accessible remotely out of the box, however do be aware that if these interfaces are then unsecured any other local user will be able to control and administer the WildFly installation.

Remoting Subsystem

The Remoting subsystem exposes a connector to allow for inbound comunications with JNDI and the EJB subsystem by default we associate the `ApplicationRealm` with this connection.

```
<subsystem xmlns="urn:jboss:domain:remoting:3.0">
  <endpoint worker="default"/>
  <http-connector name="http-remoting-connector" connector-ref="default"
security-realm="ApplicationRealm"/>
</subsystem>
```

Outbound Connections

Remoting Subsystem

Outbound connections can also be defined within the Remoting subsystem, these are typically used for remote EJB invocations from one AS server to another, in this scenario the security realm is used to obtain the server identity either it's password for X.509 certificate and possibly a trust store to verify the certificate of the remote host.

Even if the referenced realm contains username and password authentication configuration the client side of the connection will NOT use this to verify the remote server.

```
<remote-outbound-connection name="remote-ejb-connection"
                            outbound-socket-binding-ref="binding-remote-ejb-connection"
                            username="user1"
                            security-realm="PasswordRealm">
```

The security realm is only used to obtain the password for this example, as you can see here the username is specified separately.

Slave Host Controller

When running in domain mode slave host controllers need to establish a connection to the native interface of the master domain controller so these also need a realm for the identity of the slave.

```
<domain-controller>
  <remote host="${jboss.domain.master.address}" port="${jboss.domain.master.port:9999}"
security-realm="ManagementRealm"/>
</domain-controller>
```

By default when a slave host controller authenticates against the master domain controller it uses its configured name as its username. If you want to override the username used for authentication a `username` attribute can be added to the `<remote />` element.

Authentication

One of the primary functions of the security realms is to define the user stores that will be used to verify the identity of inbound connections, the actual approach taken at the transport level is based on the capabilities of these backing store definitions. The security realms are used to secure inbound connections for both the http management interface and for inbound remoting connections for both the native management interface and to access other services exposed over remoting - because of this there are some small differences between how the realm is used for each of these.

At the transport level we support the following authentication mechanisms.

HTTP	Remoting (SASL)
None	Anonymous
N/A	JBoss Local User
Digest	Digest
Basic	Plain
Client Cert	Client Cert

The most notable are the first two in this list as they need some additional explanation - the final 3 are fairly standard mechanisms.

If either the http interface, the native interface or a remoting connection are difined **without** a security realm reference then they are effectively unsecured, in the case of the http interface this means that no authentication will be performed on the incoming connection - for the remoting connections however we make use of SASL so require at least one authentication mechanism so make use of the anonymous mechanism to allow a user in without requiring a validated authentication process.

The next mechanism 'JBoss Local User' is specific to the remoting connections - as we ship WildFly secured by default we wanted a way to allow users to connect to their own AS installation after it is started without mandating that they define a user with a password - to accomplish this we have added the 'JBoss Local User' mechanism. This mechanism makes the use of tokens exchanged on the filesystem to prove that the client is local to the AS installation and has the appropriate file permissions to read a token written by the AS to file. As this mechanism is dependent on both server and client implementation details it is only supported for the remoting connections and not the http connections - at some point we may review if we can add support for this to the http interface but we would need to explore the options available with the commony used web browsers that are used to communicate with the http interface.

The Digest mechanism is simply the HTTP Digest / SASL Digest mechanism that authenticates the user by making use of md5 hashed including nonces to avoid sending passwords in plain text over the network - this is the preferred mechanism for username / password authentication.

The HTTP Basic / SASL Plain mechanism is made available for times that Digest can not be used but effectively this means that the users password will be sent over the network in the clear unless SSL is enabled.

The final mechanism Client-Cert allows X.509 certificates to be used to verify the identity of the remote client.

One point bearing in mind is that it is possible that an association with a realm can mean that a single incoming connection has the ability to choose between one or more authentication mechanisms. As an example it is possible that an incoming remoting connection could choose between 'Client Cert', A username password mechanism or 'JBoss Local User' for authentication - this would allow say a local user to use the local mechanism, a remote user to supply their username and password whilst a remote script could make a call and authenticate using a certificate.

Authorization

The actual security realms are not involved in any authorization decisions. However, they can be configured to load a user's roles, which will subsequently be used to make authorization decisions - when references to authorization are seen in the context of security realms, it is this loading of roles that is being referred to.

For the loading of roles, the process is split out to occur after the authentication step so after a user has been authenticated, a second step will occur to load the roles based on the username they used to authenticate with.

Out Of The Box Configuration

Before describing the complete set of configuration options available within the realms, we will look at the default configuration, as for most users, that is going to be the starting point before customising further.

The examples here are taken from the standalone configuration. However, the descriptions are equally applicable to domain mode. One point worth noting is that all security realms defined in the `host.xml` are available to be referenced within the domain configuration for the servers running on that host controller.

Management Realm

```
<security-realm name="ManagementRealm">
  <authentication>
    <local default-user="$local"/>
    <properties path="mgmt-users.properties" relative-to="jboss.server.config.dir"/>
  </authentication>
</security-realm>
```

The realm `ManagementRealm` is the simplest realm within the default configuration. This realm simply enables two authentication mechanisms, the local mechanism and username/password authentication which will be using Digest authentication.

- local

When using the local mechanism, it is optional for remote clients to send a username to the server. This configuration specifies that where clients do not send a username, it will be assumed that the clients username is `$local` - the `<local />` element can also be configured to allow other usernames to be specified by remote clients. However, for the default configuration, this is not enabled so is not supported.

- properties

For username / password authentication the users details will be loaded from the file `mgmt-users.properties` which is located in `{jboss.home}/standalone/configuration` or `{jboss.home}/domain/configuration` depending on the running mode of the server.

Each user is represented on their own line and the format of each line is `username=HASH` where `HASH` is a pre-prepared hash of the users password along with their username and the name of the realm which in this case is `ManagementRealm`.

You do not need to worry about generating the entries within the properties file as we provide a utility `add-user.sh` or `add-user.bat` to add the users, this utility is described in more detail below.

By pre-hashing the passwords in the properties file it does mean that if the user has used the same password on different realms then the contents of the file falling into the wrong hands does not nescesarily mean all accounts are compromised. **HOWEVER** the contents of the files do still need to be protected as they can be used to access any server where the realm name is the same and the user has the same username and password pair.

Application Realm

```
<security-realm name="ApplicationRealm">
  <authentication>
    <local default-user="$local" allowed-users="*"/>
    <properties path="application-users.properties" relative-to="jboss.server.config.dir"/>
  </authentication>
  <authorization>
    <properties path="application-roles.properties" relative-to="jboss.server.config.dir"/>
  </authorization>
</security-realm>
```

The realm `ApplicationRealm` is a slightly more complex realm as this is used for both

Authentication

The authentication configuration is very similar to the `ManagementRealm` in that it enabled both the local mechanism and a username/password based Digest mechanism.

- local

The local configuration is similar to the `ManagementRealm` in that where the remote user does not supply a username it will be assumed that the username is `$local`, however in addition to this there is now an `allowed-users` attribute with a value of `'*'` - this means that the remote user can specify any username and it will be accepted over the local mechanism provided that the local verification is a success.

To restrict the usernames that can be specified by the remote user a comma separated list of usernames can be specified instead within the `allowed-users` attribute.

- properties

The properties definition works in exactly the same way as the definition for `ManagementRealm` except now the properties file is called `application-users.properties`.

Authorization

The contents of the `Authorization` element are specific to the `ApplicationRealm`, in this case a properties file is used to load a users roles.

The properties file is called `application-roles.properties` and is located in `{jboss.home}/standalone/configuration` or `{jboss.home}/domain/configuration` depending on the running mode of the server. The format of this file is `username=ROLES` where *ROLES* is a comma separated list of the users roles.

As the loading of a users roles is a second step this is where it may be desirable to restrict which users can use the local mechanism so that some users still require username and password authentication for their roles to be loaded.

other security domain

```
<security-domain name="other" cache-type="default">
  <authentication>
    <login-module code="Remoting" flag="optional">
      <module-option name="password-stacking" value="useFirstPass"/>
    </login-module>
    <login-module code="RealmDirect" flag="required">
      <module-option name="password-stacking" value="useFirstPass"/>
    </login-module>
  </authentication>
</security-domain>
```

When applications are deployed to the application server they are associated with a security domain within the security subsystem, the `other` security domain is provided to work with the `ApplicationRealm`, this domain is defined with a pair of login modules Remoting and RealmDirect.

- Remoting

The `Remoting` login module is used to check if the request currently being authenticated is a request received over a Remoting connection, if so the identity that was created during the authentication process is used and associated with the current request.

If the request did not arrive over a Remoting connection this module does nothing and allows the JAAS based login to continue to the next module.

- RealmDirect

The `RealmDirect` login module makes use of a security realm to authenticate the current request if that did not occur in the `Remoting` login module and then use the realm to load the users roles, by default this login module assumes the realm to use is called `ApplicationRealm` although other names can be overridden using the "realm" module-option.

The advantage of this approach is that all of the backing store configuration can be left within the realm with the security domain just delegating to the realm.

user.sh

For use with the default configuration we supply a utility `add-user` which can be used to manage the properties files for the default realms used to store the users and their roles.

The add-user utility can be used to manage both the users in the `ManagementRealm` and the users in the `ApplicationRealm`, changes made apply to the properties file used both for domain mode and standalone mode.

After you have installed your application server and decided if you are going to run in standalone mode or domain mode you can delete the parent folder for the mode you are not using, the add-user utility will then only be managing the properties file for the mode in use.

The add-user utility is a command line utility however it can be run in both interactive and non-interactive mode. Depending on your platform the script to run the add-user utility is either `add-user.sh` or `add-user.bat` which can be found in `{jboss.home}/bin`.

This guide now contains a couple of examples of this utility in use to accomplish the most common tasks.

Adding a User

Adding users to the properties files is the primary purpose of this utility. Usernames can only contain the following characters in any number and in any order:

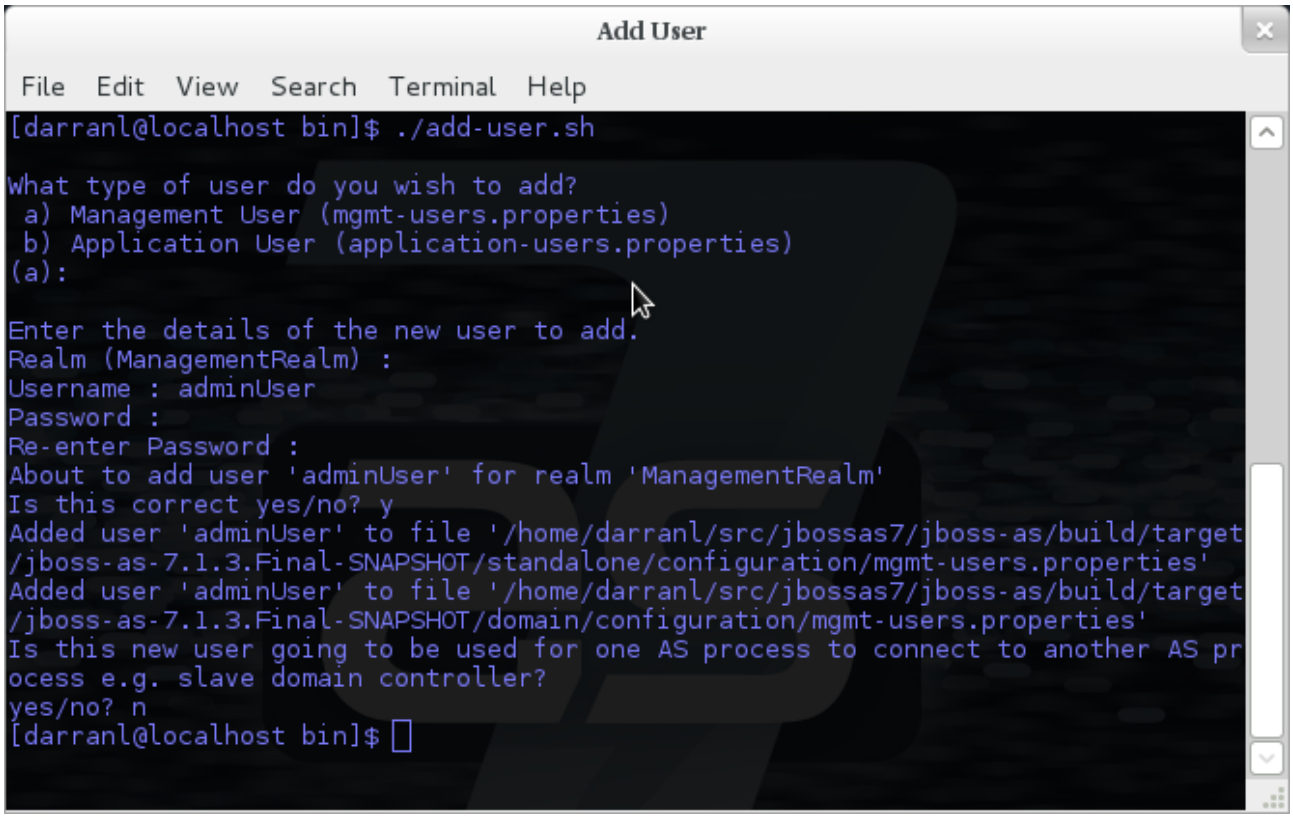
- Alphanumeric characters (a-z, A-Z, 0-9)
- Dashes (-), periods (.), commas (,), at (@)
- Escaped backslash (\\ )
- Escaped equals (\=)

The server caches the contents of the properties files in memory, however the server does check the modified time of the properties files on each authentication request and re-load if the time has been updated - this means all changes made by this utility are immediately applied to any running server.

A Management User

The default name of the realm for management users is `ManagementRealm`, when the utility prompts for the realm name just accept the default unless you have switched to a different realm.

Interactive Mode

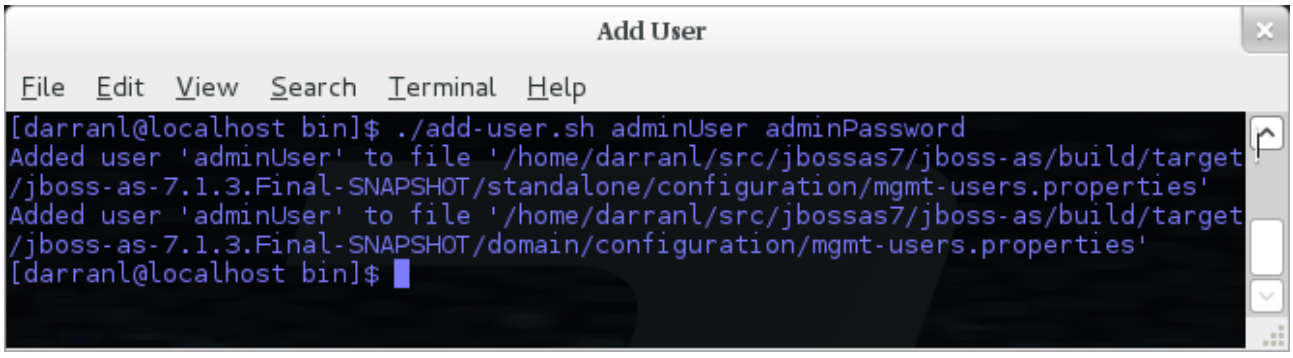


Here we have added a new Management User called `adminUser`, as you can see some of the questions offer default responses so you can just press enter without repeating the default value.

For now just answer n or no to the final question, adding users to be used by processes is described in more detail in the domain management chapter.

Interactive Mode

To add a user in non-interactive mode the command `./add-user.sh {username} {password}` can be used.

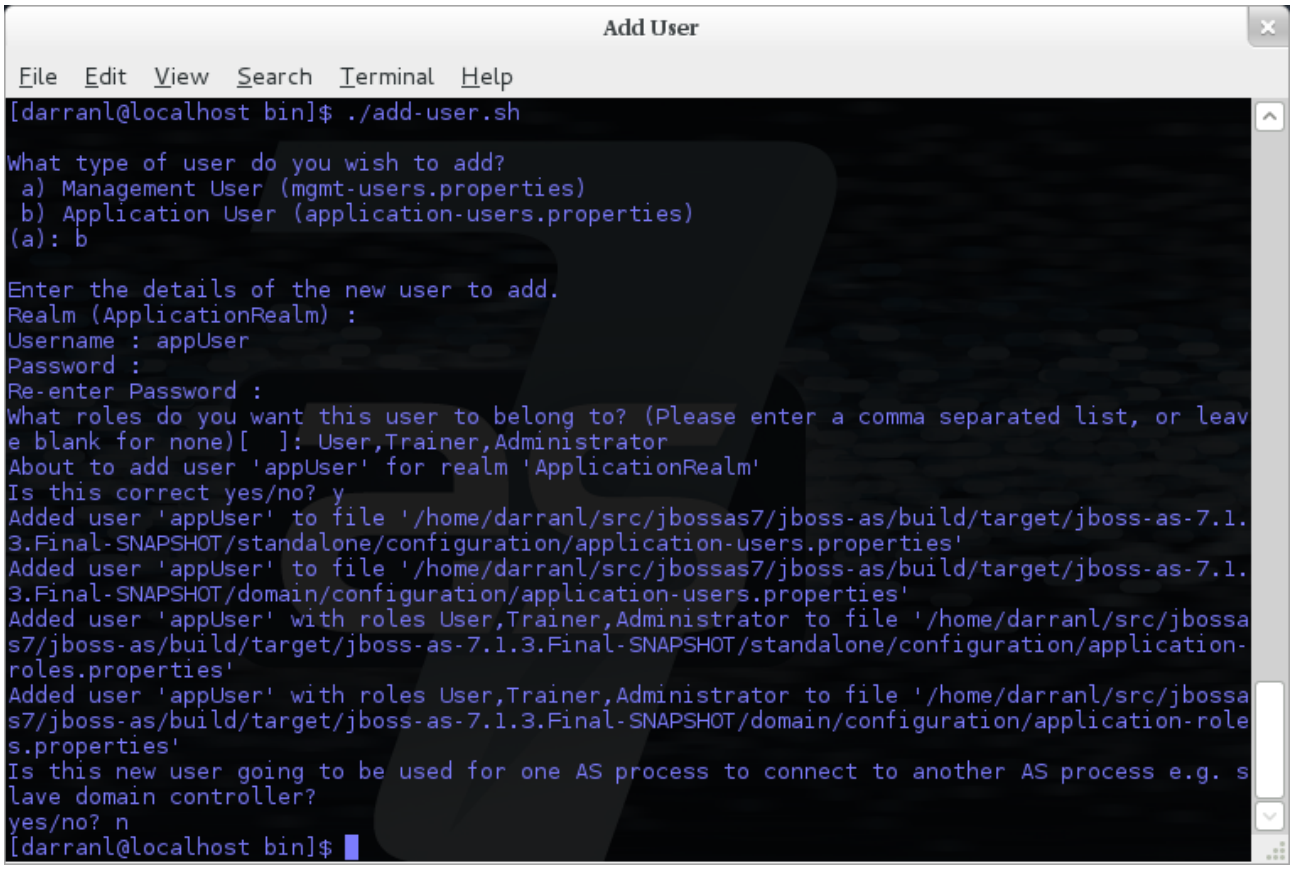


If you add users using this approach there is a risk that any other user that can view the list of running process may see the arguments including the password of the user being added, there is also the risk that the username / password combination will be cached in the history file of the shell you are currently using.

An Application User

When adding application users in addition to adding the user with their pre-hashed password it is also now possible to define the roles of the user.

Interactive Mode

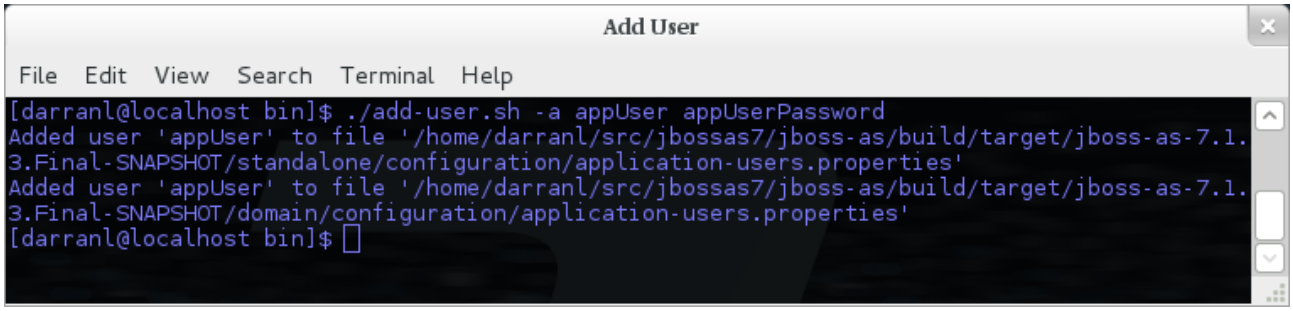


Here a new user called `appUser` has been added, in this case a comma separated list of roles has also been specified.

As with adding a management user just answer `n` or `no` to the final question until you know you are adding a user that will be establishing a connection from one server to another.

Interactive Mode

To add an application user non-interactively use the command `./add-user.sh -a {username} {password}`.



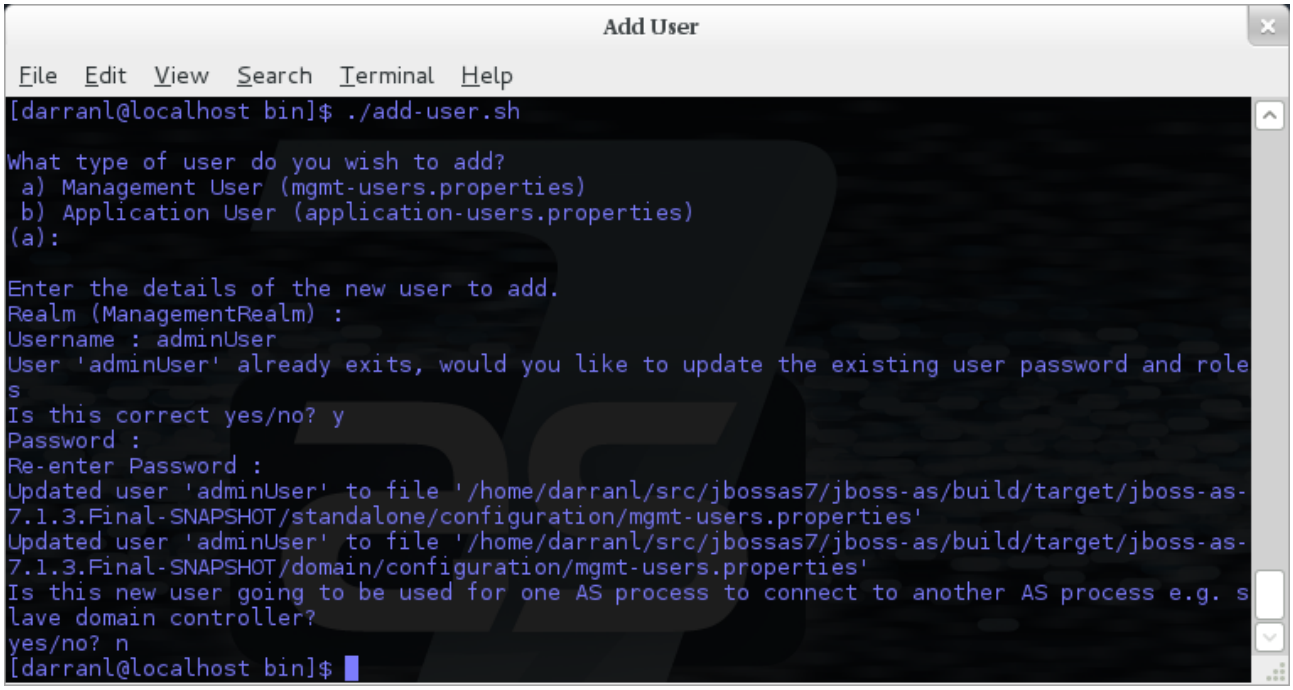
> ⚠ Non-interactive mode does not support defining a list of users, to associate a user with a set of roles you will need to manually edit the `application-roles.properties` file by hand.

Updating a User

Within the add-user utility it is also possible to update existing users, in interactive mode you will be prompted to confirm if this is your intention.

A Management User

Interactive Mode

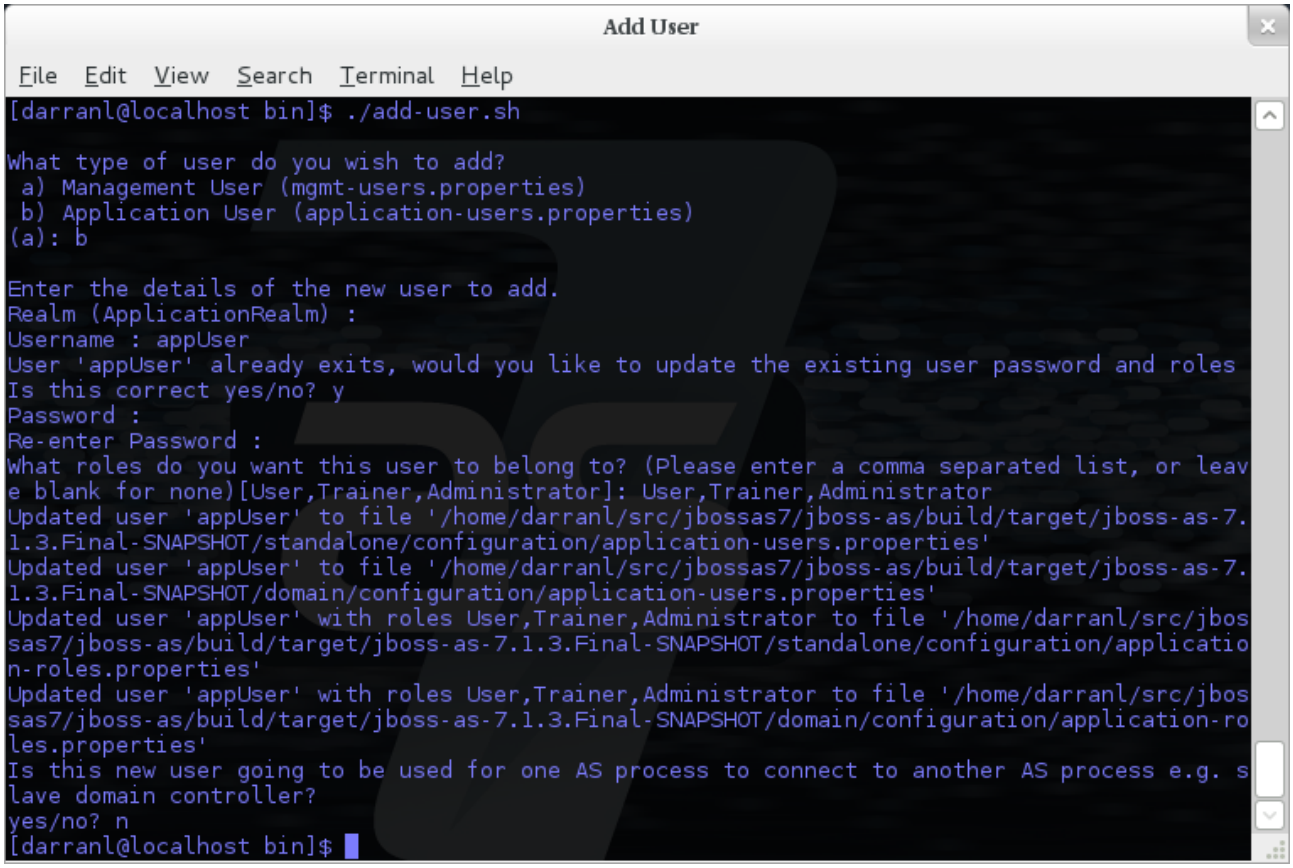


Interactive Mode

In non-interactive mode if a user already exists the update is automatic with no confirmation prompt.

An Application User

Interactive Mode



On updating a user with roles you will need to re-enter the list of roles assigned to the user.

Interactive Mode

In non-interactive mode if a user already exists the update is automatic with no confirmation prompt.

Community Contributions

There are still a few features to add to the add-user utility such as removing users or adding application users with roles in non-interactive mode, if you are interested in contributing to WildFly development the add-user utility is a good place to start as it is a stand alone utility, however it is a part of the AS build so you can become familiar with the AS development processes without needing to delve straight into the internals of the application server.

JMX Security

When configuring the security realms remote access to the server's MBeanServer needs a special mention. When running in standalone mode the following is the default configuration:

```
<subsystem xmlns="urn:jboss:domain:jmx:1.3">
  ...
  <remoting-connector/>
</subsystem>
```

With this configuration remote access to JMX is provided over the http management interface, this is secured using the realm `ManagementRealm`, this means that any user that can connect to the native interface can also use this interface to access the MBeanServer - to disable this just remove the `<remoting-connector />` element.

In domain mode it is slightly more complicates as the native interface is exposed by the host controller process however each application server is running in it's own process so by default remote access to JMX is disabled.

```
<subsystem xmlns="urn:jboss:domain:remoting:3.0">
  <http-connector name="http-remoting-connector" connector-ref="default"
security-realm="ApplicationRealm"/>
</subsystem>
```

```
<subsystem xmlns="urn:jboss:domain:jmx:1.3">
  ...
  <!--<remoting-connector use-management-endpoint="false"/>-->
</subsystem>
```

To enable remote access to JMX uncomment the `<remoting-connector />` element however be aware that this will make the MBeanServer accessible over the same Remoting connector used for remote JNDI and EJB access - this means that any user that can authenticate against the realm `ApplicationRealm` will be able to access the MBeanServer.

The following Jira issue is currently outstanding to allow access to the individual MBeanServers by proxying through the host controllers native interface AS7-4009, if this is a feature you would use please add your vote to the issue.

Detailed Configuration

This section of the documentation describes the various configuration options when defining realms, plug-ins are a slightly special case so the configuration options for plug-ins is within it's own section.

Within a security realm definition there are four optional elements `<plug-ins />`, `<server-identities />`, `<authentication />`, and `<authorization />`, as mentioned above plug-ins is defined within it's own section below so we will begin by looking at the `<server-identities />` element.

<server-identities />

The server identities section of a realm definition is used to define how a server appears to the outside world, currently this element can be used to configure a password to be used when establishing a remote outbound connection and also how to load a X.509 key which can be used for both inbound and outbound SSL connections.

<ssl />

```
<server-identities>
  <ssl protocol="...">
    <keystore path="..." relative-to="..." keystore-password="..." alias="..."
key-password="..." />
  </ssl>
</server-identities>
```

- **protocol** - By default this is set to TLS and in general does not need to be set.

The SSL element then contains the nested `<keystore />` element, this is used to define how to load the key from the file based (JKS) keystore.

- **path** (mandatory) - This is the path to the keystore, this can be an absolute path or relative to the next attribute.
- **relative-to** (optional) - The name of a service representing a path the keystore is relative to.
- **keystore-password** (mandatory) - The password required to open the keystore.
- **alias** (optional) - The alias of the entry to use from the keystore - for a keystore with multiple entries in practice the first usable entry is used but this should not be relied on and the alias should be set to guarantee which entry is used.
- **key-password** (optional) - The password to load the key entry, if omitted the keystore-password will be used instead.

If you see the error `UnrecoverableKeyException: Cannot recover key` the most likely cause that you need to specify a `key-password` and possible even an `alias` as well to ensure only one key is loaded.

<secret />

```
<server-identities>
  <secret value="..." />
</server-identities>
```

- **value** (mandatory) - The password to use for outbound connections encoded as Base64, this field also supports a vault expression should stronger protection be required.

The username for the outbound connection is specified at the point the outbound connection is defined.

<authentication />

The authentication element is predominantly used to configure the authentication that is performed on an inbound connection, however there is one exception and that is if a trust store is defined - on negotiating an outbound SSL connection the trust store will be used to verify the remote server.

```
<authentication>
  <truststore />
  <local />
  <jaas />
  <ldap />
  <properties />
  <users />
  <plug-in />
</authentication>
```

An authentication definition can have zero or one `<truststore />`, it can also have zero or one `<local />` and it can also have one of `<jaas />`, `<ldap />`, `<properties />`, `<users />`, and `<plug-in />` i.e. the local mechanism and a truststore for certificate verification can be independent switched on and off and a single username / password store can be defined.

<truststore />

```
<authentication>
  <truststore path="..." relative-to="..." keystore-password="..."/>
</authentication>
```

This element is used to define how to load a key store file that can be used as the trust store within the SSLContext we create internally, the store is then used to verify the certificates of the remote side of the connection be that inbound or outbound.

- **path** (mandatory) - This is the path to the keystore, this can be an absolute path or relative to the next attribute.
- **relative-to** (optional) - The name of a service representing a path the keystore is relative to.
- **keystore-password** (mandatory) - The password required to open the keystore.

Although this is a definition of a trust store the attribute for the password is `keystore-password`, this is because the underlying file being opened is still a key store.

<local />

```
<authentication>
  <local default-user="..." allowed-users="..." />
</authentication>
```

This element switches on the local authentication mechanism that allows clients to the server to verify that they are local to the server, at the protocol level it is optional for the remote client to send a user name in the authentication response.

- **default-user** (optional) - If the client does not pass in a username this is the assumed username, this value is also automatically added to the list of allowed-users.
- **allowed-users** (optional) - This attribute is used to specify a comma separated list of users allowed to authenticate using the local mechanism, alternatively '*' can be specified to allow any username to be specified.

<jaas />

```
<authentication>
  <jaas name="..." />
</authentication>
```

The jaas element is used to enable username and password based authentication where the supplied username and password are verified by making use of a configured jaas domain.

- **name** (mandatory) - The name of the jaas domain to use to verify the supplied username and password.

As JAAS authentication works by taking a username and password and verifying these the use of this element means that at the transport level authentication will be forced to send the password in plain text, any interception of the messages exchanged between the client and server without SSL enabled will reveal the users password.

<ldap />

```
<authentication>
  <ldap connection="..." base-dn="..." recursive="..." user-dn="...">
    <username-filter attribute="..." />
    <advanced-filter filter="..." />
  </ldap>
</authentication>
```

The ldap element is used to define how LDAP searches will be used to authenticate a user, this works by first connecting to LDAP and performing a search using the supplied user name to identity the distinguished name of the user and then a subsequent connection is made to the server using the password supplied by the user - if this second connection is a success then authentication succeeds.

Due to the verification approach used this configuration causes the authentication mechanisms selected for the protocol to cause the password to be sent from the client in plain text, the following Jira issue is to investigating proxying a Digest authentication with the LDAP server so no plain text password is needed AS7-4195.

- **connection** (mandatory) - The name of the connection to use to connect to LDAP.
- **base-dn** (mandatory) - The distinguished name of the context to use to begin the search from.
- **recursive** (optional) - Should the filter be executed recursively? Defaults to false.
- **user-dn** (optional) - After the user has been found specifies which attribute to read for the users distinguished name, defaults to 'dn'.

Within the ldap element only one of `<username-filter />` or `<advanced-filter />` can be specified.

<username-filter />
This element is used for a simple filter to match the username specified by the remote user against a single attribute, as an example with Active Directory the match is most likely to be against the '`sAMAccountName`' attribute.

- **attribute** (mandatory) - The name of the field to match the users supplied username against.

<advanced-filter />
This element is used where a more advanced filter is required, one example use of this filter is to exclude certain matches by specifying some additional criteria for the filter.

- **filter** (mandatory) - The filter to execute to locate the user, this filter should contain '`{0}`' as a place holder for the username supplied by the user authenticating.

<properties />

```
<authentication>
  <properties path="..." relative-to="..." plain-text="..." />
</authentication>
```

The properties element is used to reference a properties file to load to read a users password or pre-prepared digest for the authentication process.

- **path** (mandatory) - The path to the properties file, either absolute or relative to the path referenced by the relative-to attribute.
- **relative-to** (optional) - The name of a path service that the defined path will be relative to.
- **plain-text** (optional) - Setting to specify if the passwords are stored as plain text within the properties file, defaults to false.

By default the properties files are expected to store a pre-prepared hash of the users password in the form HEX(MD5(username ':' realm ':' password))

<users />

```
<authentication>
  <users>
    <user username="...">
      <password>...</password>
    </user>
  </users>
</authentication>
```

This is a very simple store of a username and password that stores both of these within the domain model, this is only really provided for the provision of simple examples.

- **username** (mandatory) - A users username.

The `<password/>` element is then used to define the password for the user.

<authorization />

The authorization element is used to define how a users roles can be loaded after the authentication process completes, these roles may then be used for subsequent authorization decisions based on the service being accessed. At the moment only a properties file approach or a custom plug-in are supported - support for loading roles from LDAP or from a database are planned for a subsequent release.

```
<authorization>
  <properties />
  <plug-in />
</authorization>
```

<properties />

```
<authorization>
  <properties path="..." relative-to="..." />
</authorization>
```

The format of the properties file is `username={ROLES}` where {ROLES} is a comma separated list of the users roles.

- **path** (mandatory) - The path to the properties file, either absolute or relative to the path referenced by the relative-to attribute.
- **relative-to** (optional) - The name of a path service that the defined path will be relative to.

<outbound-connection />

Strictly speaking these are not a part of the security realm definition, however at the moment they are only used by security realms so the definition of outbound connection is described here.

```
<management>
  <security-realms />
  <outbound-connections>
    <ldap />
  </outbound-connections>
</management>
```

<ldap />

At the moment we only support outbound connections to ldap servers for the authentication process - this will later be expanded when we add support for database based authentication.

```
<outbound-connections>
  <ldap name="..." url="..." search-dn="..." search-credential="..."
initial-context-factory="..." />
</outbound-connections>
```

The outbound connections are defined in this section and then referenced by name from the configuration that makes use of them.

- **name** (mandatory) - The unique name used to reference this connection.
- **url** (mandatory) - The URL use to establish the LDAP connection.
- **search-dn** (mandatory) - The distinguished name of the user to authenticate as to perform the searches.
- **search-credential** (mandatory) - The password required to connect to LDAP as the search-dn.
- **initial-context-factory** (optional) - Allows overriding the initial context factory, defaults to '`com.sun.jndi.ldap.LdapCtxFactory`'

Plug Ins

Within WildFly 8 for communication with the management interfaces and for other services exposed using Remoting where username / password authentication is used the use of Digest authentication is preferred over the use of HTTP Basic or SASL Plain so that we can avoid the sending of password in the clear over the network. For validation of the digests to work on the server we either need to be able to retrieve a users plain text password or we need to be able to obtain a ready prepared hash of their password along with the username and realm.

Previously to allow the addition of custom user stores we have added an option to the realms to call out to a JAAS domain to validate a users username and password, the problem with this approach is that to call JAAS we need the remote user to send in their plain text username and password so that a JAAS LoginModule can perform the validation, this forces us down to use either the HTTP Basic authentication mechanism or the SASL Plain mechanism depending on the transport used which is undesirable as we can not longer use Digest.

To overcome this we now support plugging in custom user stores to support loading a users password, hash and roles from a custom store to allow different stores to be implemented without forcing the authentication back to plain text variant, this article describes the requirements for a plug in and shows a simple example plug-in for use with WildFly 8.

When implementing a plug in there are two steps to the authentication process, the first step is to load the users identity and credential from the relevant store - this is then used to verify the user attempting to connect is valid. After the remote user is validated we then load the users roles in a second step. For this reason the support for plug-ins is split into the two stages, when providing a plug-in either of these two steps can be implemented but there is no requirement to implement the other side.

When implementing a plug-in the following interfaces are the bare minimum that need to be implemented so depending on if a plug-in to load a users identity or a plug-in to load a users roles is being implemented you will be implementing one of these interfaces.

***Note** - All classes and interfaces of the SPI to be implemented are in the 'org.jboss.as.domain.management.plugin' package which is a part of the 'org.jboss.as.domain-management' module but for simplicity for the rest of this section only the short names will be shown.*

AuthenticationPlugIn

To implement an `AuthenticationPlugIn` the following interface needs to be implemened: -

```
public interface AuthenticationPlugIn<T extends Credential> {
    Identity<T> loadIdentity(final String userName, final String realm) throws IOException;
}
```

During the authentication process this method will be called with the user name supplied by the remote user and the name of the realm they are authenticating against, this method call represents that an authentication attempt is occurring but it is the Identity instance that is returned that will be used for the actual authentication to verify the remote user.

The Identity interface is also an interface you will implement: -

```
public interface Identity<T extends Credential> {
    String getUserName();
    T getCredential();
}
```

Additional information can be contained within the Identity implementation although it will not currently be used, the key piece of information here is the Credential that will be returned - this needs to be one of the following: -

PasswordCredential

```
public final class PasswordCredential implements Credential {
    public PasswordCredential(final char[] password);
    public char[] getPassword();
    void clear();
}
```

The `PasswordCredential` is already implemented so use this class if you have the plain text password of the remote user, by using this the secured interfaces will be able to continue using the Digest mechanism for authentication.

DigestCredential

```
public final class DigestCredential implements Credential {
    public DigestCredential(final String hash);
    public String getHash();
}
```

This class is also already implemented and should be returned if instead of the plain text password you already have a pre-prepared hash of the username, realm and password.

ValidatePasswordCredential

```
public interface ValidatePasswordCredential extends Credential {
    boolean validatePassword(final char[] password);
}
```

This is a special Credential type to use when it is not possible to obtain either a plain text representation of the password or a pre-prepared hash - this is an interface as you will need to provide an implementation to verify a supplied password. The down side of using this type of Credential is that the authentication mechanism used at the transport level will need to drop down from Digest to either HTTP Basic or SASL Plain which will now mean that the remote client is sending their credential across the network in the clear.

If you use this type of credential be sure to force the mechanism choice to Plain as described in the configuration section below.

AuthorizationPlugIn

If you are implementing a custom mechanism to load a users roles you need to implement the `AuthorizationPlugIn`

```
public interface AuthorizationPlugIn {
    String[] loadRoles(final String userName, final String realm) throws IOException;
}
```

As with the `AuthenticationPlugIn` this has a single method that takes a users userName and realm - the return type is an array of Strings with each entry representing a role the user is a member of.

PlugInConfigurationSupport

In addition to the specific interfaces above there is an additional interface that a plug-in can implement to receive configuration information before the plug-in is used and also to receive a Map instance that can be used to share state between the plug-in instance used for the authentication step of the call and the plug-in instance used for the authorization step.

```
public interface PlugInConfigurationSupport {
    void init(final Map<String, String> configuration, final Map<String, Object> sharedState)
throws IOException;
}
```

Installing and Configuring a Plug-In

The next step of this article describes the steps to implement a plug-in provider and how to make it available within WildFly 8 and how to configure it. Example configuration and an example implementation are shown to illustrate this.

The following is an example security realm definition which will be used to illustrate this: -

```
<security-realm name="PlugInRealm">
   <plug-ins>
      <plug-in module="org.jboss.as.sample.plugin"/>
   </plug-ins>
   <authentication>
      <plug-in name="Sample">
         <properties>
            <property name="darranl.password" value="dpd"/>
            <property name="darranl.roles" value="Admin,Banker,User"/>
         </properties>
      </plug-in>
   </authentication>
   <authorization>
      <plug-in name="Delegate" />
   </authorization>
</security-realm>
```

Before looking closely at the packaging and configuration there is one more interface to implement and that is the `PlugInProvider` interface, that interface is responsible for making PlugIn instances available at runtime to handle the requests.

PlugInProvider

```
public interface PlugInProvider {
    AuthenticationPlugIn<Credential> loadAuthenticationPlugIn(final String name);
    AuthorizationPlugIn loadAuthorizationPlugIn(final String name);
}
```

These methods are called with the name that is supplied in the plug-in elements that are contained within the authentication and authorization elements of the configuration, based on the sample configuration above the loadAuthenticationPlugIn method will be called with a parameter of 'Sample' and the loadAuthorizationPlugIn method will be called with a parameter of 'Delegate'.

Multiple plug-in providers may be available to the application server so if a `PlugInProvider` implementation does not recognise a name then it should just return null and the server will continue searching the other providers. If a `PlugInProvider` does recognise a name but fails to instantiate the PlugIn then a `RuntimeException` can be thrown to indicate the failure.

As a server could have many providers registered it is recommended that a naming convention including some form of hierarchy is used e.g. use package style names to avoid conflicts.

For the example the implementation is as follows: -

```
public class SamplePluginProvider implements PlugInProvider {

    public AuthenticationPlugIn<Credential> loadAuthenticationPlugIn(String name) {
        if ("Sample".equals(name)) {
            return new SampleAuthenticationPlugIn();
        }
        return null;
    }

    public AuthorizationPlugIn loadAuthorizationPlugIn(String name) {
        if ("Sample".equals(name)) {
            return new SampleAuthenticationPlugIn();
        } else if ("Delegate".equals(name)) {
            return new DelegateAuthorizationPlugIn();
        }
        return null;
    }
}
```

The load methods are called for each authentication attempt but it will be an implementation detail of the provider if it decides to return a new instance of the provider each time - in this scenario as we also use configuration and shared state then new instances of the implementations make sense.

To load the provider use a ServiceLoader so within the META-INF/services folder of the jar this project adds a file called '`org.jboss.as.domain.management.plugin.PlugInProvider`' - this contains a single entry which is the fully qualified class name of the PlugInProvider implementation class.

```
org.jboss.as.sample.SamplePluginProvider
```

Package as a Module

To make the `PlugInProvider` available to the application it is bundled as a module and added to the modules already shipped with WildFly 8.

To add as a module we first need a `module.xml`: -

```
<?xml version="1.0" encoding="UTF-8"?>

<module xmlns="urn:jboss:module:1.1" name="org.jboss.as.sample.plugin">
    <properties>
    </properties>

    <resources>
        <resource-root path="SamplePlugIn.jar"/>
    </resources>

    <dependencies>
        <module name="org.jboss.as.domain-management" />
    </dependencies>
</module>
```

The interfaces being implemented are in the '`org.jboss.as.domain-management`' module so a dependency on that module is defined, this `module.xml` is then placed in the '{ `jboss.home}/modules/org/jboss/as/sample/plugin/main`'.

The compiled classed and `META-INF/services` as described above are assembled into a jar called `SamplePlugIn.jar` and also placed into this folder.

Looking back at the sample configuration at the top of the realm definition the following element was added: -

```
<plug-ins>
      <plug-in module="org.jboss.as.sample.plugin"/>
   </plug-ins>
```

This element is used to list the modules that should be searched for plug-ins. As plug-ins are loaded during the server start up this search is a lazy search so don't expect a definition to a non existant module or to a module that does not contain a plug-in to report an error.

The AuthenticationPlugIn

The example `AuthenticationPlugIn` is implemented as: -

```
public class SampleAuthenticationPlugIn extends AbstractPlugIn {

    private static final String PASSWORD_SUFFIX = ".password";
    private static final String ROLES_SUFFIX = ".roles";
    private Map<String, String> configuration;

    public void init(Map<String, String> configuration, Map<String, Object> sharedState) throws
IOException {
        this.configuration = configuration;
        // This will allow an AuthorizationPlugIn to delegate back to this instance.
        sharedState.put(AuthorizationPlugIn.class.getName(), this);
    }

    public Identity loadIdentity(String userName, String realm) throws IOException {
        String passwordKey = userName + PASSWORD_SUFFIX;
        if (configuration.containsKey(passwordKey)) {
            return new SampleIdentity(userName, configuration.get(passwordKey));
        }
        throw new IOException("Identity not found.");
    }

    public String[] loadRoles(String userName, String realm) throws IOException {
        String rolesKey = userName + ROLES_SUFFIX;
        if (configuration.containsKey(rolesKey)) {
            String roles = configuration.get(rolesKey);
            return roles.split(",");
        } else {
            return new String[0];
        }
    }

    private static class SampleIdentity implements Identity {
        private final String userName;
        private final Credential credential;

        private SampleIdentity(final String userName, final String password) {
            this.userName = userName;
            this.credential = new PasswordCredential(password.toCharArray());
        }

        public String getUserName() {
            return userName;
        }

        public Credential getCredential() {
            return credential;
        }
    }
}
```

As you can see from this implementation there is also an additional class being extended `AbstractPlugIn` - that is simply an abstract class that implements the `AuthenticationPlugIn`, `AuthorizationPlugIn`, and `PlugInConfigurationSupport` interfaces already. The properties that were defined in the configuration are passed in as a Map and importantly for this sample the plug-in adds itself to the shared state map.

The AuthorizationPlugIn

The example implementation of the authentication plug in is as follows: -

```
public class DelegateAuthorizationPlugIn extends AbstractPlugIn {

    private AuthorizationPlugIn authorizationPlugIn;

    public void init(Map<String, String> configuration, Map<String, Object> sharedState) throws
IOException {
        authorizationPlugIn = (AuthorizationPlugIn)
sharedState.get(AuthorizationPlugIn.class.getName());
    }

    public String[] loadRoles(String userName, String realm) throws IOException {
        return authorizationPlugIn.loadRoles(userName, realm);
    }

}
```

This plug-in illustrates how two plug-ins can work together, by the `AuthenticationPlugIn` placing itself in the shared state map it is possible for the authorization plug-in to make use of it for the loadRoles implementation.

Another option to consider to achieve similar behaviour could be to provide an Identity implementation that also contains the roles and place this in the shared state map - the `AuthorizationPlugIn` can retrieve this and return the roles.

Forcing Plain Text Authentication

As mentioned earlier in this article if the `ValidatePasswordCredential` is going to be used then the authentication used at the transport level needs to be forced from Digest authentication to plain text authentication, this can be achieved by adding a mechanism attribute to the plug-in definition within the authentication element i.e.

```
<authentication>
    <plug-in name="Sample" mechanism="PLAIN">
```

Example Configurations

This section of the document contains a couple of examples for the most common scenarios likely to be used with the security realms, please feel free to raise Jira issues requesting additional scenarios or if you have configured something not covered here please feel free to add your own examples - this document is editable after all

At the moment these examples are making use of the '`ManagementRealm`' however the same can apply to the '`ApplicationRealm`' or any custom realm you create for yourselves.

LDAP Authentication

The following example demonstrates an example configuration making use of Active Directory to verify the users username and password.

```
<management>
  <security-realms>
    <security-realm name="ManagementRealm">
      <authentication>
        <ldap connection="EC2" base-dn="CN=Users,DC=darranl,DC=jboss,DC=org">
          <username-filter attribute="sAMAccountName" />
        </ldap>
      </authentication>
    </security-realm>

  </security-realms>

  <outbound-connections>
    <ldap name="EC2" url="ldap://127.0.0.1:9797"
search-dn="CN=wf8,CN=Users,DC=darranl,DC=jboss,DC=org" search-credential="password"/>
  </outbound-connections>

  ...

</management>
```

For simplicity the `<local/>` configuration has been removed from this example, however there it is fine to leave that in place for local authentication to remain possible.

Enable SSL

The first step is the creation of the key, by default this is going to be used for both the native management interface and the http management interface - to create the key we can use the `keyTool`, the following example will create a key valid for one year.

Open a terminal window in the folder `{jboss.home}/standalone/configuration` and enter the following command: -

```
keytool -genkey -alias server -keyalg RSA -keystore server.keystore -validity
365
```

```
Enter keystore password:
Re-enter new password:
```

In this example I choose '`keystore_password`'.

```
What is your first and last name?
  [Unknown]:  localhost
```

Of all of the questions asked this is the most important and should match the host name that will be entered into the web browser to connect to the admin console.

Answer the remaining questions as you see fit and at the end for the purpose of this example I set the key password to '`key_password`'.

The following example shows how this newly created keystore will be referenced to enable SSL.

```
<security-realm name="ManagementRealm">
  <server-identities>
    <ssl>
      <keystore path="server.keystore" relative-to="jboss.server.config.dir"
keystore-password="keystore_password" alias="server" key-password="key_password" />
    </ssl>
  </server-identities>
  <authentication>
    ...
  </authentication>
</security-realm>
```

The contents of the `<authentication />` have not been changed in this example so authentication still occurs using either the local mechanism or username/password authentication using Digest.

Add Client-Cert to SSL

To enable Client-Cert style authentication we just now need to add a `<truststore />` element to the `<authentication />` element referencing a trust store that has had the certificates or trusted clients imported.

```
<security-realm name="ManagementRealm">
  <server-identities>
    <ssl>
      <keystore path="server.keystore" relative-to="jboss.server.config.dir"
keystore-password="keystore_password" alias="server" key-password="key_password" />
    </ssl>
  </server-identities>
  <authentication>
    <truststore path="server.truststore" relative-to="jboss.server.config.dir"
keystore-password="truststore_password" />
    <local default-user="$local"/>
    <properties path="mgmt-users.properties" relative-to="jboss.server.config.dir"/>
  </authentication>
</security-realm>
```

In this scenario if Client-Cert authentication does not occur clients can fall back to use either the local mechanism or username/password authentication. To make Client-Cert based authentication mandatory just remove the `<local />` and `<properties />` elements.

5.5.2 Authorizing management actions with Role Based Access Control

WildFly introduces a Role Based Access Control scheme that allows different administrative users to have different sets of permissions to read and update parts of the management tree. This replaces the simple permission scheme used in JBoss AS 7, where anyone who could successfully authenticate to the management security realm would have all permissions.

Access Control Providers

WildFly ships with two access control "providers", the "simple" provider, and the "rbac" provider. The "simple" provider is the default, and provides a permission scheme equivalent to the JBoss AS 7 behavior where any authenticated administrator has all permissions. The "rbac" provider gives the finer grained permission scheme that is the focus of this section.

The access control configuration is included in the management section of a standalone server's standalone.xml, or in a new "management" section in a managed domain's domain.xml. The access control policy is centrally configured in a managed domain.

```
<management>
    . . .
    <access-control provider="simple">
        <role-mapping>
            <role name="SuperUser">
                <include>
                    <user name="$local"/>
                </include>
            </role>
        </role-mapping>
    </access-control>
</management>
```

As you can see, the provider is set to "simple" by default. With the "simple" provider, the nested "role-mapping" section is not actually relevant. It's there to help ensure that if the provider attribute is switched to "rbac" there will be at least one user mapped to a role that can continue to administer the system. This default mapping assigns the "$local" user name to the RBAC role that provides all permissions, the "SuperUser" role. The "$local" user name is the name an administrator will be assigned if he or she uses the CLI on the same system as the WildFly instance and the "local" authentication scheme is enabled.

RBAC provider overview

The access control scheme implemented by the "rbac" provider is based on seven standard roles. A role is a named set of permissions to perform one of the actions: addressing (i.e. looking up) a management resource, reading it, or modifying it. The different roles have constraints applied to their permissions that are used to determine whether the permission is granted.

RBAC roles

The seven standard roles are divided into two broad categories, based on whether the role can deal with items that are considered to be "security sensitive". Resources, attributes and operations that may affect administrative security (e.g. security realm resources and attributes that contain passwords) are "security sensitive".

Four roles are not given permissions for "security sensitive" items:

- Monitor – a read-only role. Cannot modify any resource.
- Operator – Monitor permissions, plus can modify runtime state, but cannot modify anything that ends up in the persistent configuration. Could, for example, restart a server.
- Maintainer – Operator permissions, plus can modify the persistent configuration.
- Deployer – like a Maintainer, but with permission to modify persistent configuration constrained to resources that are considered to be "application resources". A deployment is an application resource. The messaging server is not. Items like datasources and JMS destinations are not considered to be application resources by default, but this is configurable.

Three roles are granted permissions for security sensitive items:

- SuperUser – has all permissions. Equivalent to a JBoss AS 7 administrator.
- Administrator – has all permissions except cannot read or write resources related to the administrative audit logging system.
- Auditor – can read anything. Can only modify the resources related to the administrative audit logging system.

The Auditor and Administrator roles are meant for organizations that want a separation of responsibilities between those who audit normal administrative actions and those who perform them, with those who perform most actions (Administrator role) not being able to read or alter the auditing configuration.

Access control constraints

The following factors are used to determine whether a given role is granted a permission:

- What the requested action is (address, read, write)
- Whether the resource, attribute or operation affects the persistent configuration
- Whether the resource, attribute or operation is related to the administrative audit logging function
- Whether the resource, attribute or operation is configured as security sensitive
- Whether an attribute or operation parameter value has a security vault expression
- Whether a resource is considered to be associated with applications, as opposed to being part of a general container configuration

The first three of these factors are non-configurable; the latter three allow some customization. See "Configuring constraints" for details.

Addressing a resource

As mentioned above, permissions are granted to perform one of three actions, addressing a resource, reading it, and modifying. The latter two actions are fairly self-explanatory. But what is meant by "addressing" a resource?

"Addressing" a resource refers to taking an action that allows the user to determine whether a resource at a given address actually exists. For example, the "read-children-names" operation lets a user determine valid addresses. Trying to read a resource and getting a "Permission denied" error also gives the user a clue that there actually is a resource at the requested address.

Some resources may include sensitive information as part of their address. For example, security realm resources include the realm name as the last element in the address. That realm name is potentially security sensitive; for example it is part of the data used when creating a hash of a user password. Because some addresses may contain security sensitive data, a user needs permission to even "address" a resource. If a user attempts to address a resource and does not have permission, they will not receive a "permission denied" type error. Rather, the system will respond as if the resource does not even exist, e.g. excluding the resource from the result of the "read-children-names" operation or responding with a "No such resource" error instead of "Permission denied" if the user is attempting to read or write the resource.

Another term for "addressing" a resource is "looking up" the resource.

Switching to the "rbac" provider

Use the CLI to switch the access-control provider.

Before changing the provider to "rbac", be sure your configuration has a user who will be mapped to one of the RBAC roles, preferably with at least one in the Administrator or SuperUser role. Otherwise your installation will not be manageable except by shutting it down and editing the xml configuration. If you have started with one of the standard xml configurations shipped with WildFly, the "$local" user will be mapped to the "SuperUser" role and the "local" authentication scheme will be enabled. This will allow a user running the CLI on the same system as the WildFly process to have full administrative permissions. Remote CLI users and web-based admin console users will have no permissions.

We recommend mapping at least one user besides "$local" before switching the provider to "rbac". You can do all of the configuration associated with the "rbac" provider even when the provider is set to "simple"

The management resources related to access control are located in the `core-service=management/access=authorization` portion of the management resource tree. Update the `provider` attribute to change between the "simple" and "rbac" providers. Any update requires a reload or restart to take effect.

```
[standalone@localhost:9990 /] cd core-service=management/access=authorization
[standalone@localhost:9990 access=authorization] :write-attribute(name=provider,value=rbac)
{
    "outcome" => "success",
    "response-headers" => {
        "operation-requires-reload" => true,
        "process-state" => "reload-required"
    }
}
[standalone@localhost:9990 access=authorization] reload
```

In a managed domain, the access control configuration is part of the domain wide configuration, so the resource address is the same as above, but the CLI is connected to the master Domain Controller:

```
[domain@localhost:9990 /] cd core-service=management/access=authorization
[domain@localhost:9990 access=authorization] :write-attribute(name=provider,value=rbac)
{
    "outcome" => "success",
    "response-headers" => {
        "operation-requires-reload" => true,
        "process-state" => "reload-required"
    },
    "result" => undefined,
    "server-groups" => {"main-server-group" => {"host" => {"master" => {
        "server-one" => {"response" => {
            "outcome" => "success",
            "response-headers" => {
                "operation-requires-reload" => true,
                "process-state" => "reload-required"
            }
        }},
        "server-two" => {"response" => {
            "outcome" => "success",
            "response-headers" => {
                "operation-requires-reload" => true,
                "process-state" => "reload-required"
            }
        }}
    }}}}
}
[domain@localhost:9990 access=authorization] reload --host=master
```

As with a standalone server, a reload or restart is required for the change to take effect. In this case, all hosts and servers in the domain will need to be reloaded or restarted, starting with the master Domain Controller, so be sure to plan well before making this change.

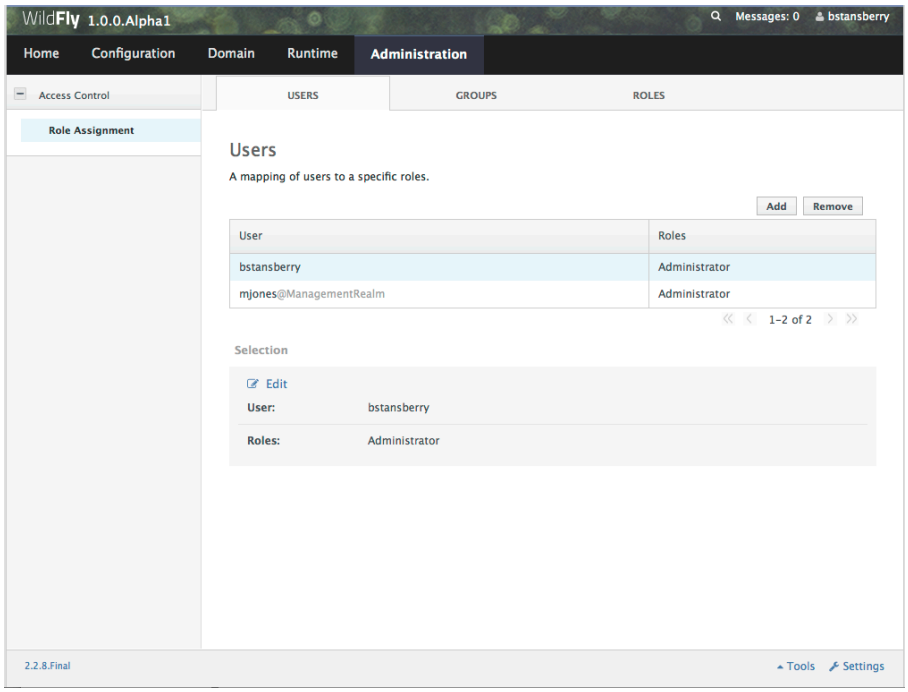
Mapping users and groups to roles

Once the "rbac" access control provider is enabled, only users who are mapped to one of the available roles will have any administrative permissions at all. So, to make RBAC useful, a mapping between individual users or groups of users and the available roles must be performed.

Mapping individual users

The easiest way to map individual users to roles is to use the web-based admin console.

Navigate to the "Administration" tab and the "Users" subtab. From there individual user mappings can be added, removed, or edited.



The CLI can also be used to map individuals users to roles.

First, if one does not exist, create the parent resource for all mappings for a role. Here we create the resource for the `Administrator` role.

```
[domain@localhost:9990 /]
/core-service=management/access=authorization/role-mapping=Administrator:add
{
    "outcome" => "success",
    "result" => undefined,
    "server-groups" => {"main-server-group" => {"host" => {"master" => {
        "server-one" => {"response" => {"outcome" => "success"}},
        "server-two" => {"response" => {"outcome" => "success"}}
    }}}}
}
```

Once this is done, map a user to the role:

```
[domain@localhost:9990 /]
/core-service=management/access=authorization/role-mapping=Administrator/include=user-jsmith:add(n
"outcome" => "success",
    "result" => undefined,
    "server-groups" => {"main-server-group" => {"host" => {"master" => {
        "server-one" => {"response" => {"outcome" => "success"}},
        "server-two" => {"response" => {"outcome" => "success"}}
    }}}}
}
```

Now if user `jsmith` authenticates to any security realm associated with the management interface they are using, he will be mapped to the `Administrator` role.

To restrict the mapping to a particular security realm, change the `realm` attribute to the realm name. This might be useful if different realms are associated with different management interfaces, and the goal is to limit a user to a particular interface.

```
[domain@localhost:9990 /]
/core-service=management/access=authorization/role-mapping=Administrator/include=user-mjones:add(n
"outcome" => "success",
    "result" => undefined,
    "server-groups" => {"main-server-group" => {"host" => {"master" => {
        "server-one" => {"response" => {"outcome" => "success"}},
        "server-two" => {"response" => {"outcome" => "success"}}
    }}}}
}
```

User groups

A "group" is an arbitrary collection of users that may exist in the end user environment. They can be named whatever the end user organization wants and can contain whatever users the end user organization wants. Some of the authentication store types supported by WildFly security realms include the ability to access information about what groups a user is a member of and associate this information with the `Subject` produced when the user is authenticated. This is currently supported for the following authentication store types:

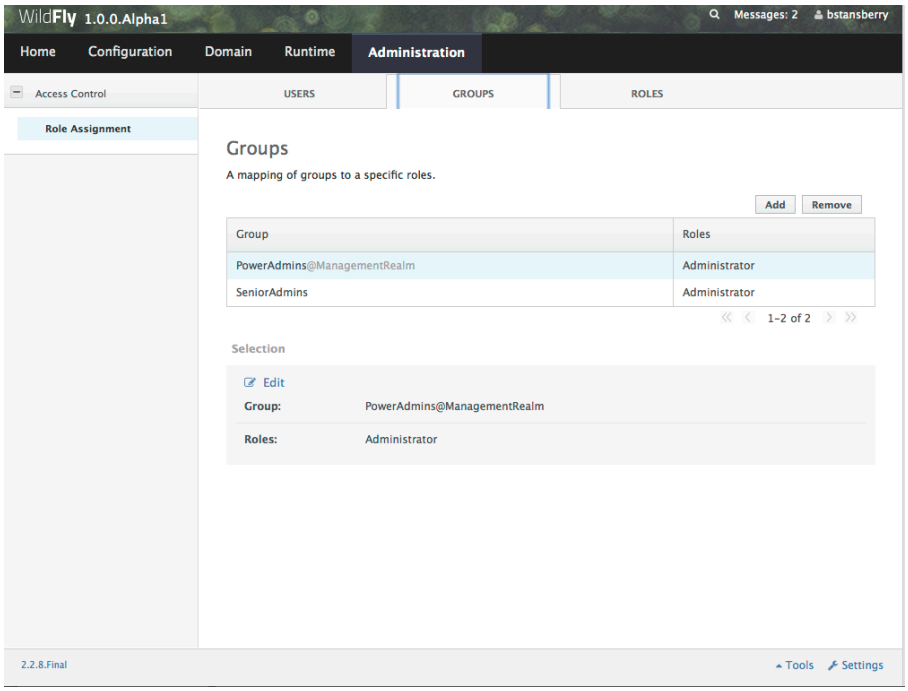
- properties file (via the `<realm_name>-groups.properties` file)
- LDAP (via directory-server-specific configuration)

Groups are convenient when it comes to associating a user with a role, since entire groups can be associated with a role in a single mapping.

Mapping groups to roles

The easiest way to map groups to roles is to use the web-based admin console.

Navigate to the "Administration" tab and the "Groups" subtab. From there group mappings can be added, removed, or edited.



The CLI can also be used to map groups to roles. The only difference to individual user mapping is the value of the `type` attribute should be `GROUP` instead of `USER`.

```
[domain@localhost:9990 /]
/core-service=management/access=authorization/role-mapping=Administrator/include=group-SeniorAdmin
"outcome" => "success",
    "result" => undefined,
    "server-groups" => {"main-server-group" => {"host" => {"master" => {
        "server-one" => {"response" => {"outcome" => "success"}},
        "server-two" => {"response" => {"outcome" => "success"}}
    }}}}
}
```

As with individual user mappings, the mapping can be restricted to users authenticating via a particular security realm:

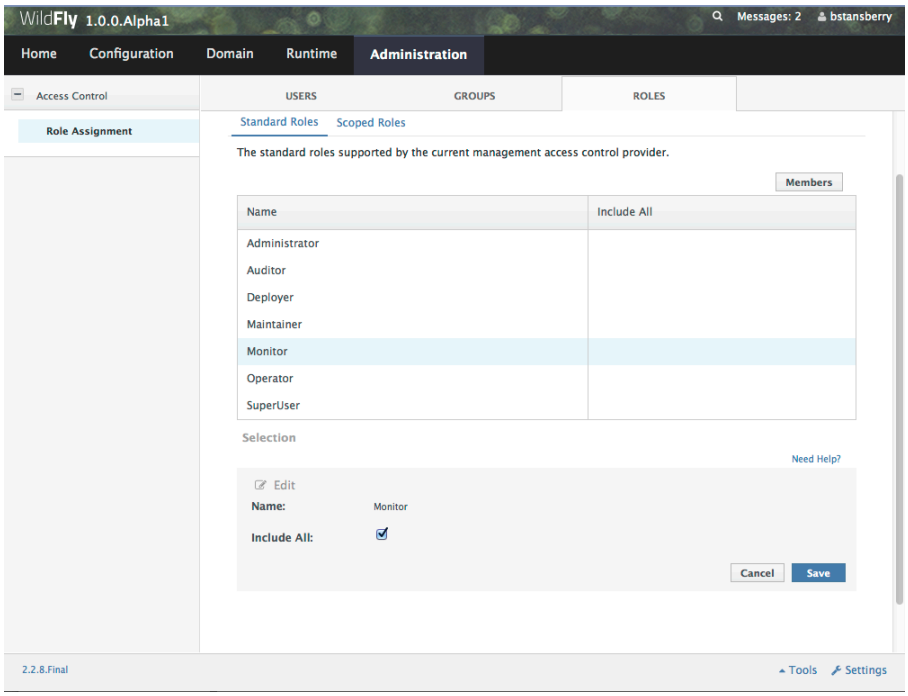
```
[domain@localhost:9990 /]
/core-service=management/access=authorization/role-mapping=Administrator/include=group-PowerAdmins
"outcome" => "success",
    "result" => undefined,
    "server-groups" => {"main-server-group" => {"host" => {"master" => {
        "server-one" => {"response" => {"outcome" => "success"}},
        "server-two" => {"response" => {"outcome" => "success"}}
    }}}}
}
```

Including all authenticated users in a role

It's possible to specify that all authenticated users should be mapped to a particular role. This could be used, for example, to ensure that anyone who can authenticate can at least have `Monitor` privileges.

> ⚠ A user who can authenticate to the management security realm but who does not map to a role will not be able to perform any administrative functions, not even reads.

In the web based admin console, navigate to the "Administration" tab, "Roles" subtab, highlight the relevant role, click the "Edit" button and click on the "Include All" checkbox:



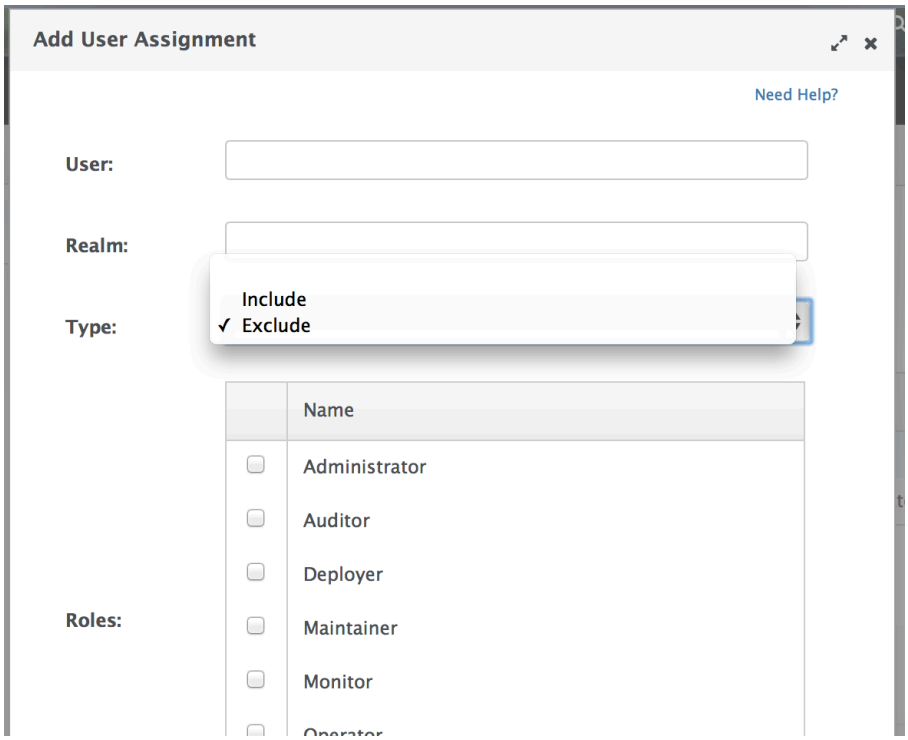
The same change can be made using the CLI:

```
[domain@localhost:9990 /]
/core-service=management/access=authorization/role-mapping=Monitor:write-attribute(name=include-al
"outcome" => "success",
    "result" => undefined,
    "server-groups" => {"main-server-group" => {"host" => {"master" => {
        "server-one" => {"response" => {"outcome" => "success"}},
        "server-two" => {"response" => {"outcome" => "success"}}
    }}}}
}
```

Excluding users and groups

It is also possible to explicitly exclude certain users and groups from a role. Exclusions take precedence over inclusions, including cases where the `include-all` attribute is set to true for a role.

In the admin console, excludes are done in the same screens as includes. In the add dialog, simply change the "Type" pulldown to "Exclude".



In the CLI, excludes are identical to includes, except the resource address has `exclude` instead of `include` as the key for the last address element:

```
[domain@localhost:9990 /]
/core-service=management/access=authorization/role-mapping=Monitor/exclude=group-Temps:add(name=Te
"outcome" => "success",
    "result" => undefined,
    "server-groups" => {"main-server-group" => {"host" => {"master" => {
        "server-one" => {"response" => {"outcome" => "success"}},
        "server-two" => {"response" => {"outcome" => "success"}}
    }}}}
}
```

Users who map to multiple roles

It is possible that a given user will be mapped to more than one role. When this occurs, by default the user will be granted the union of the permissions of the two roles. This behavior can be changed **on a global basis** to instead respond to the user request with an error if this situation is detected:

```
[standalone@localhost:9990 /] cd core-service=management/access=authorization
[standalone@localhost:9990 access=authorization]
:write-attribute(name=permission-combination-policy,value=rejecting)
{"outcome" => "success"}
```

Note that no reload is required; the change takes immediate effect.

To restore the default behavior, set the value to "permissive":

```
[standalone@localhost:9990 /] cd core-service=management/access=authorization
[standalone@localhost:9990 access=authorization]
:write-attribute(name=permission-combination-policy,value=permissive)
{"outcome" => "success"}
```

Adding custom roles in a managed domain

A managed domain may involve a variety of servers running different configurations and hosting different applications. In such an environment, it is likely that there will be different teams of administrators responsible for different parts of the domain. To allow organizations to grant permissions to only parts of a domain, WildFly's RBAC scheme allows for the creation of custom "scoped roles". Scoped roles are based on the seven standard roles, but with permissions limited to a portion of the domain – either to a set of server groups or to a set of hosts.

Server group scoped roles

The privileges for a server-group scoped role are constrained to resources associated with one or more server groups. Server groups are often associated with a particular application or set of applications; organizations that have separate teams responsible for different applications may find server-group scoped roles useful.

A server-group scoped role is equivalent to the default role upon which it is based, but with privileges constrained to target resources in the resource trees rooted in the server group resources. The server-group scoped role can be configured to include privileges for the following resources trees logically related to the server group:

- Profile
- Socket Binding Group
- Deployment
- Deployment override
- Server group
- Server config
- Server

Resources in the profile, socket binding group, server config and server portions of the tree that are not logically related to a server group associated with the server-group scoped role will not be addressable by a user in that role. So, in a domain with server groups “a” and “b”, a user in a server-group scoped role that grants access to “a” will not be able to address /server-group=b. The system will treat that resource as non-existent for that user.

In addition to these privileges, users in a server-group scoped role will have non-sensitive read privileges (equivalent to the Monitor role) for resources other than those listed above.

The easiest way to create a server-group scoped role is to use the admin console. But you can also use the CLI to create a server-group scoped role.

```
[domain@localhost:9990 /]
/core-service=management/access=authorization/server-group-scoped-role=MainGroupAdmins:add(base-ro
"outcome" => "success",
    "result" => undefined,
    "server-groups" => {"main-server-group" => {"host" => {"master" => {
        "server-one" => {"response" => {"outcome" => "success"}},
        "server-two" => {"response" => {"outcome" => "success"}}
    }}}}
}
```

Once the role is created, users or groups can be mapped to it the same as with the seven standard roles.

Host scoped roles

The privileges for a host-scoped role are constrained to resources associated with one or more hosts. A user with a host-scoped role cannot modify the domain wide configuration. Organizations may use host-scoped roles to give administrators relatively broad administrative rights for a host without granting such rights across the managed domain.

A host-scoped role is equivalent to the default role upon which it is based, but with privileges constrained to target resources in the resource trees rooted in the host resources for one or more specified hosts.

In addition to these privileges, users in a host-scoped role will have non-sensitive read privileges (equivalent to the Monitor role) for domain wide resources (i.e. those not in the /host=* section of the tree.)

Resources in the /host=* portion of the tree that are unrelated to the hosts specified for the Host Scoped Role will not be visible to users in that host-scoped role. So, in a domain with hosts “a” and “b”, a user in a host-scoped role that grants access to “a” will not be able to address /host=b. The system will treat that resource as non-existent for that user.

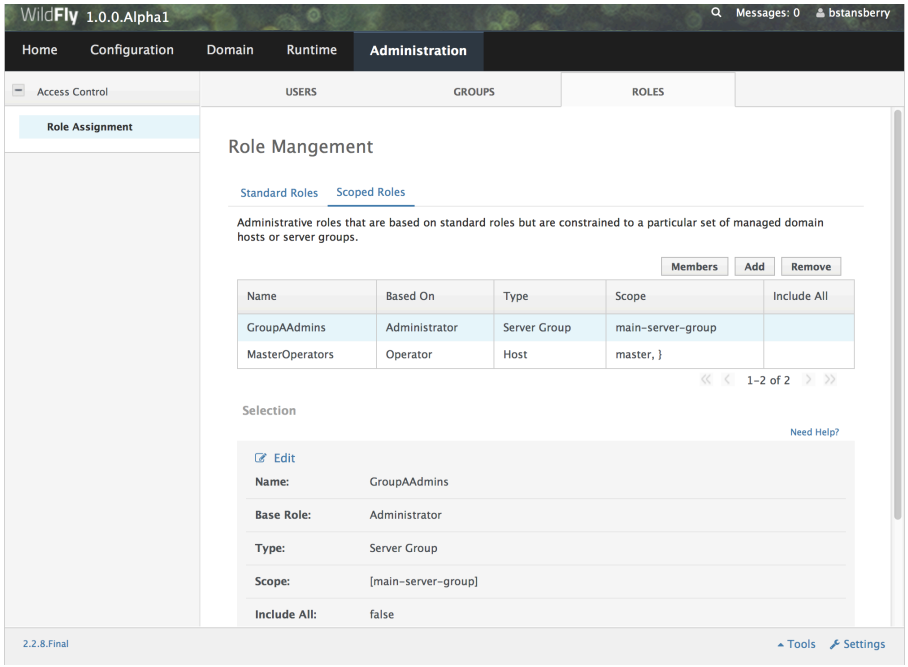
The easiest way to create a host-scoped role is to use the admin console. But you can also use the CLI to create a host scoped role.

```
[domain@localhost:9990 /]
/core-service=management/access=authorization/host-scoped-role=MasterOperators:add(base-role=Opera
"outcome" => "success",
    "result" => undefined,
    "server-groups" => {"main-server-group" => {"host" => {"master" => {
        "server-one" => {"response" => {"outcome" => "success"}},
        "server-two" => {"response" => {"outcome" => "success"}}
    }}}}
}
```

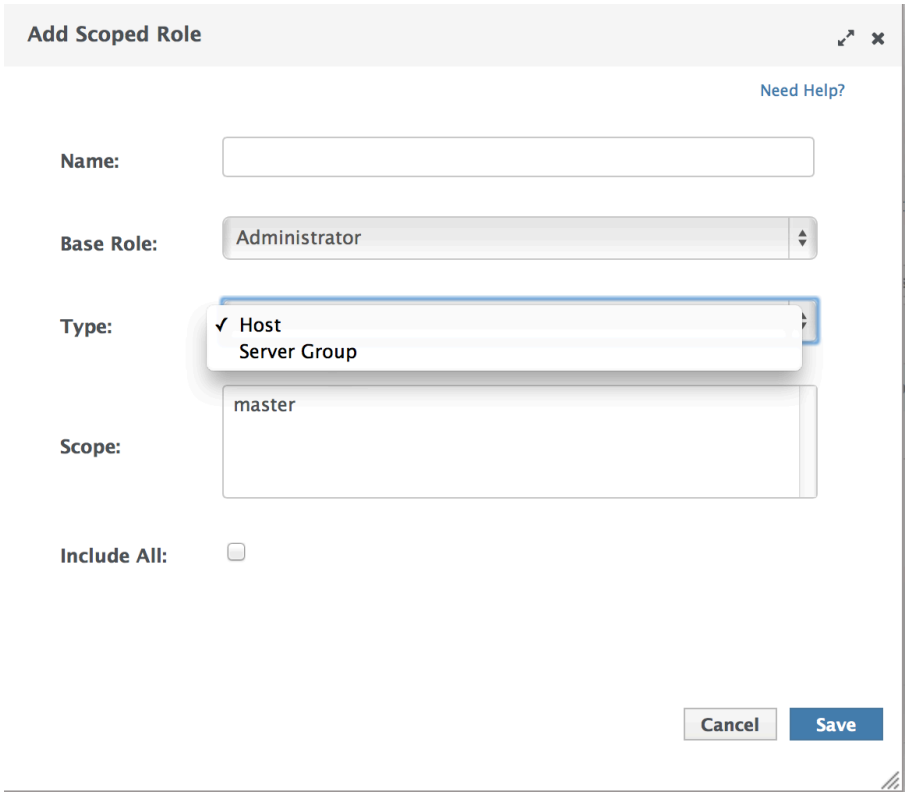
Once the role is created, users or groups can be mapped to it the same as with the seven standard roles.

Using the admin console to create scoped roles

Both server-group and host scoped roles can be added, removed or edited via the admin console. Select "Scoped Roles" from the "Administration" tab, "Roles" subtab:



When adding a new scoped role, use the dialogue's "Type" pull down to choose between a host scoped role and a server-group scoped role. Then place the names of the relevant hosts or server groups in the "Scope" text are.



Configuring constraints

The following factors are used to determine whether a given role is granted a permission:

- What the requested action is (address, read, write)
- Whether the resource, attribute or operation affects the persistent configuration
- Whether the resource, attribute or operation is related to the administrative audit logging function
- Whether the resource, attribute or operation is configured as security sensitive
- Whether an attribute or operation parameter value has a security vault expression
- Whether a resource is considered to be associated with applications, as opposed to being part of a general container configuration

The first three of these factors are non-configurable; the latter three allow some customization.

Configuring sensitivity

"Sensitivity" constraints are about restricting access to security-sensitive data. Different organizations may have different opinions about what is security sensitive, so WildFly provides configuration options to allow users to tailor these constraints.

Sensitive resources, attributes and operations

The developers of the WildFly core and of any subsystem may annotate resources, attributes or operations with a "sensitivity classification". Classifications are either provided by the core and may be applicable anywhere in the management model, or they are scoped to a particular subsystem. For each classification, there will be a setting declaring whether by default the addressing, read and write actions are considered to be sensitive. If an action is sensitive, only users in the roles able to deal with sensitive data (Administrator, Auditor, SuperUser) will have permissions.

Using the CLI, administrators can see the settings for a classification. For example, there is a core classification called "socket-config" that is applied to elements throughout the model that relate to configuring sockets:

```
[domain@localhost:9990 /] cd
core-service=management/access=authorization/constraint=sensitivity-classification/type=core/class
classification=socket-config] ls -l
ATTRIBUTE                       VALUE     TYPE
configured-requires-addressable undefined BOOLEAN
configured-requires-read        undefined BOOLEAN
configured-requires-write       undefined BOOLEAN
default-requires-addressable    false     BOOLEAN
default-requires-read           false     BOOLEAN
default-requires-write          true      BOOLEAN

CHILD      MIN-OCCURS MAX-OCCURS
applies-to n/a        n/a
```

The various `default-requires-...` attributes indicate whether a user must be in a role that allows security sensitive actions in order to perform the action. In the `socket-config` example above, `default-requires-write` is true, while the others are false. So, by default modifying a setting involving socket configuration is considered sensitive, while addressing those resources or doing reads is not sensitive.

The `default-requires-...` attributes are read-only. The `configured-requires-...` attributes however can be modified to override the default settings with ones appropriate for your organization. For example, if your organization doesn't regard modifying socket configuration settings to be security sensitive, you can change that setting:

```
[domain@localhost:9990 classification=socket-config]
:write-attribute(name=configured-requires-write,value=false)
{
    "outcome" => "success",
    "result" => undefined,
    "server-groups" => {"main-server-group" => {"host" => {"master" => {
        "server-one" => {"response" => {"outcome" => "success"}},
        "server-two" => {"response" => {"outcome" => "success"}}
    }}}}
}
```

Administrators can also read the management model to see to which resources, attributes and operations a particular sensitivity classification applies:

```
[domain@localhost:9990 classification=socket-config]
:read-children-resources(child-type=applies-to)
{
    "outcome" => "success",
    "result" => {
        "/host=master" => {
            "address" => "/host=master",
            "attributes" => [],
            "entire-resource" => false,
            "operations" => ["resolve-internet-address"]
        },
        "/host=master/core-service=host-environment" => {
            "address" => "/host=master/core-service=host-environment",
            "attributes" => [
                "host-controller-port",
                "host-controller-address",
                "process-controller-port",
                "process-controller-address"
            ],
            "entire-resource" => false,
            "operations" => []
        },
        "/host=master/core-service=management/management-interface=http-interface" => {
            "address" =>
"/host=master/core-service=management/management-interface=http-interface",
            "attributes" => [
                "port",
                "secure-interface",
```

```
                "secure-port",
                "interface"
            ],
            "entire-resource" => false,
            "operations" => []
        },
        "/host=master/core-service=management/management-interface=native-interface" => {
            "address" =>
"/host=master/core-service=management/management-interface=native-interface",
            "attributes" => [
                "port",
                "interface"
            ],
            "entire-resource" => false,
            "operations" => []
        },
        "/host=master/interface=*" => {
            "address" => "/host=master/interface=*",
            "attributes" => [],
            "entire-resource" => true,
            "operations" => ["resolve-internet-address"]
        },
        "/host=master/server-config=*/interface=*" => {
            "address" => "/host=master/server-config=*/interface=*",
            "attributes" => [],
            "entire-resource" => true,
            "operations" => []
        },
        "/interface=*" => {
            "address" => "/interface=*",
            "attributes" => [],
            "entire-resource" => true,
            "operations" => []
        },
        "/profile=*/subsystem=messaging/hornetq-server=*/broadcast-group=*" => {
            "address" => "/profile=*/subsystem=messaging/hornetq-server=*/broadcast-group=*",
            "attributes" => [
                "group-address",
                "group-port",
                "local-bind-address",
                "local-bind-port"
            ],
            "entire-resource" => false,
            "operations" => []
        },
        "/profile=*/subsystem=messaging/hornetq-server=*/discovery-group=*" => {
            "address" => "/profile=*/subsystem=messaging/hornetq-server=*/discovery-group=*",
            "attributes" => [
                "group-address",
                "group-port",
                "local-bind-address"
            ],
            "entire-resource" => false,
            "operations" => []
        },
        "/profile=*/subsystem=transactions" => {
            "address" => "/profile=*/subsystem=transactions",
            "attributes" => ["process-id-socket-max-ports"],
```

```
            "entire-resource" => false,
            "operations" => []
        },
        "/server-group=*" => {
            "address" => "/server-group=*",
            "attributes" => ["socket-binding-port-offset"],
            "entire-resource" => false,
            "operations" => []
        },
        "/socket-binding-group=*" => {
            "address" => "/socket-binding-group=*",
            "attributes" => [],
            "entire-resource" => true,
            "operations" => []
        }
    }
}
```

There will be a separate child for each address to which the classification applies. The `entire-resource` attribute will be true if the classification applies to the entire resource. Otherwise, the `attributes` and `operations` attributes will include the names of attributes or operations to which the classification applies.

Classifications with broad use
Several of the core sensitivity classifications are commonly used across the management model and deserve special mention.

Name	Description
credential	An attribute whose value is some sort of credential, e.g. a password or a username. By default sensitive for both reads and writes
security-domain-ref	An attribute whose value is the name of a security domain. By default sensitive for both reads and writes
security-realm-ref	An attribute whose value is the name of a security realm. By default sensitive for both reads and writes
socket-binding-ref	An attribute whose value is the name of a socket binding. By default not sensitive for any action
socket-config	A resource, attribute or operation that somehow relates to configuring a socket. By default sensitive for writes

Values with security vault expressions

By default any attribute or operation parameter whose value includes a security vault expression will be treated as sensitive, even if no sensitivity classification applies or the classification does not treat the action as sensitive.

This setting can be **globally** changed via the CLI. There is a resource for this configuration:

```
[domain@localhost:9990 /] cd
core-service=management/access=authorization/constraint=vault-expression
[domain@localhost:9990 constraint=vault-expression] ls -l
ATTRIBUTE                 VALUE     TYPE
configured-requires-read  undefined BOOLEAN
configured-requires-write undefined BOOLEAN
default-requires-read     true      BOOLEAN
default-requires-write    true      BOOLEAN
```

The various `default-requires-...` attributes indicate whether a user must be in a role that allows security sensitive actions in order to perform the action. So, by default both reading and writing attributes whose values include vault expressions requires a user to be in one of the roles with sensitive data permissions.

The `default-requires-...` attributes are read-only. The `configured-requires-...` attributes however can be modified to override the default settings with settings appropriate for your organization. For example, if your organization doesn't regard reading vault expressions to be security sensitive, you can change that setting:

```
[domain@localhost:9990 constraint=vault-expression]
:write-attribute(name=configured-requires-read,value=false)
{
    "outcome" => "success",
    "result" => undefined,
    "server-groups" => {"main-server-group" => {"host" => {"master" => {
        "server-one" => {"response" => {"outcome" => "success"}},
        "server-two" => {"response" => {"outcome" => "success"}}
    }}}}
}
```

This vault-expression constraint overlaps somewhat with the core "credential" sensitivity classification in that the most typical uses of a vault expression are in attributes that contain a user name or password, and those will typically be annotated with the "credential" sensitivity classification. So, if you change the settings for the "credential" sensitivity classification you may also need to make a corresponding change to the vault-expression constraint settings, or your change will not have full effect.

Be aware though, that vault expressions can be used in any attribute that supports expressions, not just in credential-type attributes. So it is important to be familiar with where and how your organization uses vault expressions before changing these settings.

Configuring "Deployer" role access

The standard Deployer role has its write permissions limited to resources that are considered to be "application resources"; i.e. conceptually part of an application and not part of the general server configuration. By default, only deployment resources are considered to be application resources. However, different organizations may have different opinions on what qualifies as an application resource, so for resource types that subsystems authors consider *potentially* to be application resources, WildFly provides a configuration option to declare them as such. Such resources will be annotated with an "application classification".

For example, the mail subsystem provides such a classification:

```
[domain@localhost:9990 /] cd
/core-service=management/access=authorization/constraint=application-classification/type=mail/clas
classification=mail-session] ls -l
ATTRIBUTE              VALUE     TYPE
configured-application undefined BOOLEAN
default-application    false     BOOLEAN

CHILD      MIN-OCCURS MAX-OCCURS
applies-to n/a        n/a
```

Use `read-resource` or `read-children-resources` to see what resources have this classification applied:

```
[domain@localhost:9990 classification=mail-session]
:read-children-resources(child-type=applies-to)
{
    "outcome" => "success",
    "result" => {"/profile=*/subsystem=mail/mail-session=*" => {
        "address" => "/profile=*/subsystem=mail/mail-session=*",
        "attributes" => [],
        "entire-resource" => true,
        "operations" => []
    }}
}
```

This indicates that this classification, intuitively enough, only applies to mail subsystem mail-session resources.

To make resources with this classification writeable by users in the Deployer role, set the `configured-application` attribute to true.

```
[domain@localhost:9990 classification=mail-session]
:write-attribute(name=configured-application,value=true)
{
    "outcome" => "success",
    "result" => undefined,
    "server-groups" => {"main-server-group" => {"host" => {"master" => {
        "server-one" => {"response" => {"outcome" => "success"}},
        "server-two" => {"response" => {"outcome" => "success"}}
    }}}}
}
```

Application classifications shipped with WildFly

The subsystems shipped with the full WildFly distribution include the following application classifications:

Subsystem	Classification
datasources	data-source
datasources	jdbc-driver
datasources	xa-data-source
logging	logger
logging	logging-profile
mail	mail-session
messaging	jms-queue
messaging	jms-topic
messaging	queue
messaging	security-setting
naming	binding
resource-adapters	resource-adapter
security	security-domain

In each case the classification applies to the resources you would expect, given its name.

RBAC effect on administrator user experience

The RBAC scheme will result in reduced permissions for administrators who do not map to the SuperUser role, so this will of course have some impact on their experience when using administrative tools like the admin console and the CLI.

Admin console

The admin console takes great pains to provide a good user experience even when the user has reduced permissions. Resources the user is not permitted to see will simply not be shown, or if appropriate will be replaced in the UI with an indication that the user is not authorized. Interaction units like "Add" and "Remove" buttons and "Edit" links will be suppressed if the user has no write permissions.

CLI

The CLI is a much more unconstrained tool than the admin console is, allowing users to try to execute whatever operations they wish, so it's more likely that users who attempt to do things for which they lack necessary permissions will receive failure messages. For example, a user in the Monitor role cannot read passwords:

```
[domain@localhost:9990 /]
/profile=default/subsystem=datasources/data-source=ExampleDS:read-attribute(name=password)
{
    "outcome" => "failed",
    "result" => undefined,
    "failure-description" => "WFLYCTL0313: Unauthorized to execute operation 'read-attribute'
for resource '[
    (\"profile\" => \"default\"),
    (\"subsystem\" => \"datasources\"),
    (\"data-source\" => \"ExampleDS\")
]' -- \"WFLYCTL0332: Permission denied\"",
    "rolled-back" => true
}
```

If the user isn't even allowed to address the resource then the response would be as if the resource doesn't exist, even though it actually does:

```
[domain@localhost:9990 /]
/profile=default/subsystem=security/security-domain=other:read-resource
{
    "outcome" => "failed",
    "failure-description" => "WFLYCTL0216: Management resource '[
    (\"profile\" => \"default\"),
    (\"subsystem\" => \"security\"),
    (\"security-domain\" => \"other\")
]' not found",
    "rolled-back" => true
}
```

This prevents unauthorized users fishing for sensitive data in resource addresses by checking for "Permission denied" type failures.

Users who use the `read-resource` operation may ask for data, some of which they are allowed to see and some of which they are not. If this happens, the request will not fail, but inaccessible data will be elided and a response header will be included advising on what was not included. Here we show the effect of a Monitor trying to recursively read the security subsystem configuration:

```
[domain@localhost:9990 /] /profile=default/subsystem=security:read-resource(recursive=true)
{
    "outcome" => "success",
    "result" => {
        "deep-copy-subject-mode" => undefined,
        "security-domain" => undefined,
        "vault" => undefined
    },
    "response-headers" => {"access-control" => [{
        "absolute-address" => [
            ("profile" => "default"),
            ("subsystem" => "security")
        ],
        "relative-address" => [],
        "filtered-attributes" => ["deep-copy-subject-mode"],
        "filtered-children-types" => ["security-domain"]
    }]}
}
```

The `response-headers` section includes access control data in a list with one element per relevant resource. (In this case there's just one.) The absolute and relative address of the resource is shown, along with the fact that the value of the `deep-copy-subject-mode` attribute has been filtered (i.e. undefined is shown as the value, which may not be the real value) as well as the fact that child resources of type `security-domain` have been filtered.

Description of access control constraints in the management model metadata

The management model descriptive metadata returned from operations like `read-resource-description` and `read-operation-description` can be configured to include information describing the access control constraints relevant to the resource, This is done by using the `access-control` parameter. The output will be tailored to the caller's permissions. For example, a user who maps to the Monitor role could ask for information about a resource in the mail subsystem:

```
[domain@localhost:9990 /] cd /profile=default/subsystem=mail/mail-session=default/server=smtp
[domain@localhost:9990 server=smtp] :read-resource-description(access-control=trim-descriptions)
{
    "outcome" => "success",
    "result" => {
        "description" => undefined,
        "access-constraints" => {"application" => {"mail-session" => {"type" => "mail"}}},
        "attributes" => undefined,
        "operations" => undefined,
        "children" => {},
        "access-control" => {
            "default" => {
                "read" => true,
                "write" => false,
                "attributes" => {
                    "outbound-socket-binding-ref" => {
                        "read" => true,
                        "write" => false
                    },
                    "username" => {
                        "read" => false,
                        "write" => false
                    },
                    "tls" => {
                        "read" => true,
                        "write" => false
                    },
                    "ssl" => {
                        "read" => true,
                        "write" => false
                    },
                    "password" => {
                        "read" => false,
                        "write" => false
                    }
                }
            },
            "exceptions" => {}
        }
    }
}
```

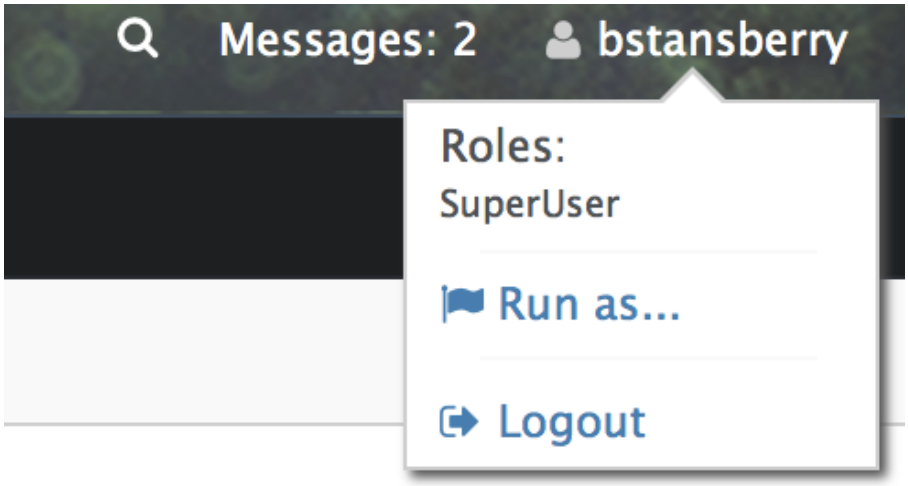
Because `trim-descriptions` was used as the value for the `access-control` parameter, the typical "description", "attributes", "operations" and "children" data is largely suppressed. (For more on this, see below.) The `access-constraints` field indicates that this resource is annotated with an [application constraint]. The `access-control` field includes information about the permissions the current caller has for this resource. The `default` section shows the default settings for resources of this type. The `read` and `write` fields directly under `default` show that the caller can, in general, read this resource but cannot write it. The `attributes` section shows the individual attribute settings. Note that Monitor cannot read the `username` and `password` attributes.

There are three valid values for the `access-control` parameter to `read-resource-description` and `read-operation-description`:

- **none** – do not include access control information in the response. This is the default behavior if no parameter is included.
- **trim-descriptions** – remove the normal description details, as shown in the example above
- **combined-descriptions** – include both the normal output and the access control data

Learning about your own role mappings

Users can learn in which roles they are operating. In the admin console, click on your name in the top right corner; the roles you are in will be shown.



CLI users should use the `whoami` operation with the `verbose` attribute set:

```
[domain@localhost:9990 /] :whoami(verbose=true)
{
    "outcome" => "success",
    "result" => {
        "identity" => {
            "username" => "aadams",
            "realm" => "ManagementRealm"
        },
        "mapped-roles" => [
            "Maintainer"
        ]
    }
}
```

"Run-as" capability for SuperUsers

If a user maps to the SuperUser role, WildFly also supports letting that user request that they instead map to one or more other roles. This can be useful when doing demos, or when the SuperUser is changing the RBAC configuration and wants to see what effect the changes have from the perspective of a user in another role. This capability is only available to the SuperUser role, so it can only be used to narrow a user's permissions, not to potentially increase them.

CLI run-as

With the CLI, run-as capability is on a per-request basis. It is done by using the "roles" operation header, the value of which can be the name of a single role or a bracket-enclosed, comma-delimited list of role names.

Example with a low level operation:

```
[standalone@localhost:9990 /] :whoami(verbose=true){roles=[Operator,Auditor]}
{
    "outcome" => "success",
    "result" => {
        "identity" => {
            "username" => "$local",
            "realm" => "ManagementRealm"
        },
        "mapped-roles" => [
            "Auditor",
            "Operator"
        ]
    }
}
```

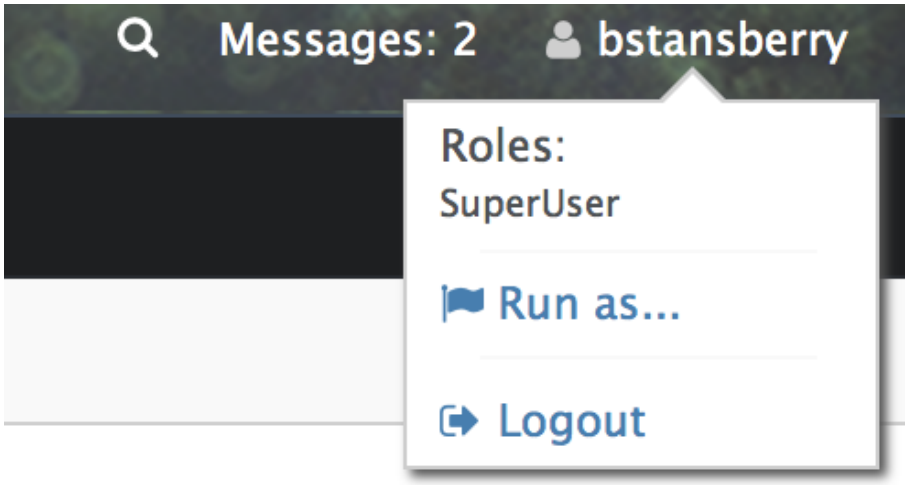
Example with a CLI command:

```
[standalone@localhost:9990 /] deploy /tmp/helloworld.war --headers={roles=Monitor}
{"WFLYCTL0062: Composite operation failed and was rolled back. Steps that failed:" =>
{"Operation step-1" => "WFLYCTL0313: Unauthorized to execute operation 'add' for resource
'[(\"deployment\" => \"helloworld.war\")]' -- \"WFLYCTL0332: Permission denied\""}}
[standalone@localhost:9990 /] deploy /tmp/helloworld.war --headers={roles=Maintainer}
```

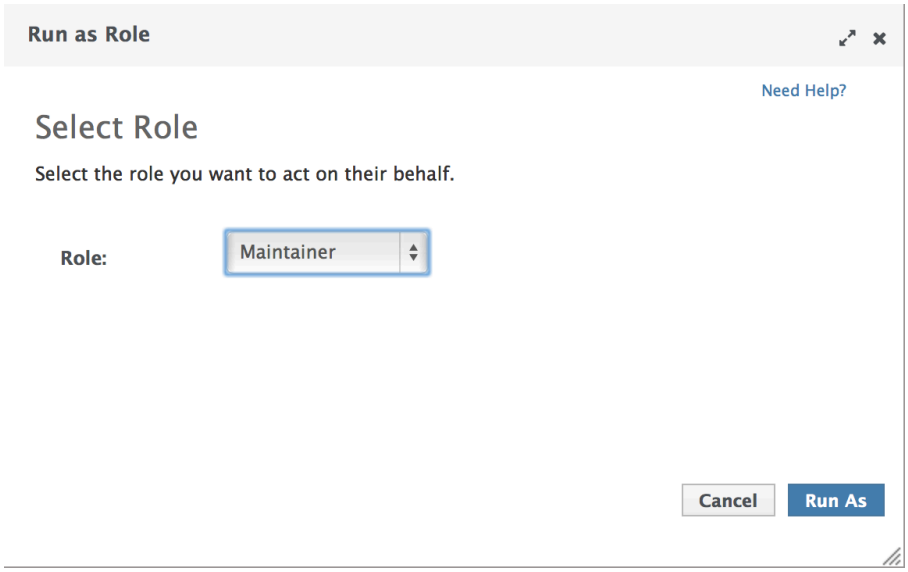
Here we show the effect of switching to a role that isn't granted the necessary permission.

Admin console run-as

Admin console users can change the role in which they operate by clicking on their name in the top right corner and clicking on the "Run as..." link.



Then select the role in which you wish to operate:



The console will need to be restarted in order for the change to take effect.

Using run-as roles with the "simple" access control provider

This "run-as" capability is available even if the "simple" access control provider is used. When the "simple" provider is used, any authenticated administrator is treated the same as if they would map to SuperUser when the "rbac" provider is used.
However, the "simple" provider actually understands all of the "rbac" provider configuration settings described above, but only makes use of them if the "run-as" capability is used for a request. Otherwise, the SuperUser role has all permissions, so detailed configuration is irrelevant.

Using the run-as capability with the "simple" provider may be useful if an administrator is setting up an rbac provider configuration before switching the provider to rbac to make that configuration take effect. The administrator can then run-as different roles to see the effect of the planned settings.

5.6 Application deployment

5.6.1 Managed Domain

In a managed domain, deployments are associated with a `server-group` (see Core management concepts). Any server within the server group will then be provided with that deployment.

The domain and host controller components manage the distribution of binaries across network boundaries.

Deployment Commands

Distributing deployment binaries involves two steps: uploading the deployment to the repository the domain controller will use to distribute its contents, and then assigning the deployment to one or more server groups.

You can do this in one sweep with the CLI:

```
[domain@localhost:9990 /] deploy ~/Desktop/test-application.war
Either --all-server-groups or --server-groups must be specified.

[domain@localhost:9990 /] deploy ~/Desktop/test-application.war --all-server-groups
'test-application.war' deployed successfully.
```

The deployment will be available to the domain controller, assigned to a server group, and deployed on all running servers in that group:

```
[domain@localhost:9990 /] :read-children-names(child-type=deployment)
{
   "outcome" => "success",
   "result" => [
       "mysql-connector-java-5.1.15.jar",
       "test-application.war"
   ]
}

[domain@localhost:9990 /]
/server-group=main-server-group/deployment=test-application.war:read-resource(include-runtime)
{
   "outcome" => "success",
   "result" => {
       "enabled" => true,
       "name" => "test-application.war",
       "managed" => true,
       "runtime-name" => "test-application.war"
   }
}
```

If you only want the deployment deployed on servers in some server groups, but not all, use the `--server-groups` parameter instead of `-all-server-groups`:

```
[domain@localhost:9990 /] deploy ~/Desktop/test-application.war
--server-groups=main-server-group,another-group
'test-application.war' deployed successfully.
```

If you have a new version of the deployment that you want to deploy replacing an existing one, use the `--force` parameter:

```
[domain@localhost:9990 /] deploy ~/Desktop/test-application.war --all-server-groups --force
'test-application.war' deployed successfully.
```

You can remove binaries from server groups with the `undeploy` command:

```
[domain@localhost:9990 /] undeploy test-application.war --all-relevant-server-groups
Successfully undeployed test-application.war.

[domain@localhost:9990 /]
/server-group=main-server-group:read-children-names(child-type=deployment)
{
   "outcome" => "success",
   "result" => []
}
```

If you only want to undeploy from some server groups but not others, use the `-server-groups` parameter instead of `-all-relevant-server-groups`.

The CLI `deploy` command supports a number of other parameters that can control behavior. Use the `--help` parameter to learn more:

```
[domain@localhost:9990 /] deploy --help
[...]
```

Managing deployments through the web interface provides an alternate, sometimes simpler approach.

Exploded managed deployments

Managed and unmanaged deployments can be 'exploded', i.e. on the filesystem in the form of a directory structure whose structure corresponds to an unzipped version of the archive. An exploded deployment can be convenient to administer if your administrative processes involve inserting or replacing files from a base version in order to create a version tailored for a particular use (for example, copy in a base deployment and then copy in a jboss-web.xml file to tailor a deployment for use in WildFly.) Exploded deployments are also nice in some development scenarios, as you can replace static content (e.g. .html, .css) files in the deployment and have the new content visible immediately without requiring a redeploy.

Since unmanaged deployment content is directly in your charge, the following operations only make sense for a managed deployment.

```
[domain@localhost:9990 /] /deployment=exploded.war:add(content=[{empty=true}])
```

This will create an empty exploded deployment to which you'll be able to add content. The **empty** content parameter is required to check that you really intend to create an empty deployment and not just forget to define the content.

```
[domain@localhost:9990 /] /deployment=kitchensink.ear:explode()
```

This will 'explode' an existing archive deployment to its exploded format. This operation is not recursive so you need to explode the sub-deployment if you want to be able to manipulate the sub-deployment content. You can do this by specifying the sub-deployment archive **path** as a parameter to the explode operation.

```
[domain@localhost:9990 /]
/deployment=kitchensink.ear:explode(path=wildfly-kitchensink-ear-web.war)
```

Now you can add or remove content to your exploded deployment. Note that per-default this will overwrite existing contents, you can specify the overwrite parameter to make the operation fail if the content already exists.

```
[domain@localhost:9990 /]
/deployment=exploded.war:add-content(content=[{target-path=WEB-INF/classes/org/jboss/as/test/deplo
input-stream-index=/home/demo/org/jboss/as/test/deployment/trivial/ServiceActivatorDeployment.clas
{target-path=META-INF/MANIFEST.MF, input-stream-index=/home/demo/META-INF/MANIFEST.MF},
{target-path=META-INF/services/org.jboss.msc.service.ServiceActivator,
input-stream-index=/home/demo/META-INF/services/org.jboss.msc.service.ServiceActivator}])
```

Each content specifies a source content and the target path to which it will be copied relative to the deployment root. With WildFly 11 you can use **input-stream-index** (which was a convenient way to pass a stream of content) from the CLI by pointing it to a local file.

```
[domain@localhost:9990 /]
/deployment=exploded.war:remove-content(paths=[WEB-INF/classes/org/jboss/as/test/deployment/trivia
META-INF/MANIFEST.MF, META-INF/services/org.jboss.msc.service.ServiceActivator])
```

Now you can list the content of an exploded deployment, or just some part of it.

```
[domain@localhost:9990 /] /deployment=kitchensink.ear:browse-content(archive=false,
path=wildfly-kitchensink-ear-web.war)
{
    "outcome" => "success",
    "result" => [
        {
            "path" => "META-INF/",
            "directory" => true
        },
        {
            "path" => "META-INF/MANIFEST.MF",
            "directory" => false,
            "file-size" => 128L
        },
        {
            "path" => "WEB-INF/",
            "directory" => true
        },
        {
            "path" => "WEB-INF/templates/",
            "directory" => true
        },
        {
            "path" => "WEB-INF/classes/",
            "directory" => true
        },
        {
            "path" => "WEB-INF/classes/org/",
            "directory" => true
        },
        {
            "path" => "WEB-INF/classes/org/jboss/",
            "directory" => true
        },
        {
```

```
            "path" => "WEB-INF/classes/org/jboss/as/",
            "directory" => true
        },
        {
            "path" => "WEB-INF/classes/org/jboss/as/quickstarts/",
            "directory" => true
        },
        {
            "path" => "WEB-INF/classes/org/jboss/as/quickstarts/kitchensink_ear/",
            "directory" => true
        },
        {
            "path" => "WEB-INF/classes/org/jboss/as/quickstarts/kitchensink_ear/controller/",
            "directory" => true
        },
        {
            "path" => "WEB-INF/classes/org/jboss/as/quickstarts/kitchensink_ear/rest/",
            "directory" => true
        },
        {
            "path" => "WEB-INF/classes/org/jboss/as/quickstarts/kitchensink_ear/util/",
            "directory" => true
        },
        {
            "path" => "resources/",
            "directory" => true
        },
        {
            "path" => "resources/css/",
            "directory" => true
        },
        {
            "path" => "resources/gfx/",
            "directory" => true
        },
        {
            "path" => "WEB-INF/templates/default.xhtml",
            "directory" => false,
            "file-size" => 2113L
        },
        {
            "path" => "WEB-INF/faces-config.xml",
            "directory" => false,
            "file-size" => 1365L
        },
        {
            "path" =>
"WEB-INF/classes/org/jboss/as/quickstarts/kitchensink_ear/controller/MemberController.class",
            "directory" => false,
            "file-size" => 2750L
        },
        {
            "path" =>
"WEB-INF/classes/org/jboss/as/quickstarts/kitchensink_ear/rest/MemberResourceRESTService.class",
            "directory" => false,
            "file-size" => 6363L
        },
        {
```

```
            "path" =>
"WEB-INF/classes/org/jboss/as/quickstarts/kitchensink_ear/rest/JaxRsActivator.class",
            "directory" => false,
            "file-size" => 464L
        },
        {
            "path" =>
"WEB-INF/classes/org/jboss/as/quickstarts/kitchensink_ear/util/WebResources.class",
            "directory" => false,
            "file-size" => 667L
        },
        {
            "path" => "WEB-INF/beans.xml",
            "directory" => false,
            "file-size" => 1262L
        },
        {
            "path" => "index.xhtml",
            "directory" => false,
            "file-size" => 3603L
        },
        {
            "path" => "index.html",
            "directory" => false,
            "file-size" => 949L
        },
        {
            "path" => "resources/css/screen.css",
            "directory" => false,
            "file-size" => 4025L
        },
        {
            "path" => "resources/gfx/headerbkg.png",
            "directory" => false,
            "file-size" => 1147L
        },
        {
            "path" => "resources/gfx/asidebkg.png",
            "directory" => false,
            "file-size" => 1374L
        },
        {
            "path" => "resources/gfx/banner.png",
            "directory" => false,
            "file-size" => 41473L
        },
        {
            "path" => "resources/gfx/bkg-blkheader.png",
            "directory" => false,
            "file-size" => 116L
        },
        {
            "path" => "resources/gfx/rhjb_eap_logo.png",
            "directory" => false,
            "file-size" => 2637L
        },
        {
            "path" => "META-INF/maven/",
```

```
            "directory" => true
        },
        {
            "path" => "META-INF/maven/org.wildfly.quickstarts/",
            "directory" => true
        },
        {
            "path" => "META-INF/maven/org.wildfly.quickstarts/wildfly-kitchensink-ear-web/",
            "directory" => true
        },
        {
            "path" =>
"META-INF/maven/org.wildfly.quickstarts/wildfly-kitchensink-ear-web/pom.xml",
            "directory" => false,
            "file-size" => 4128L
        },
        {
            "path" =>
"META-INF/maven/org.wildfly.quickstarts/wildfly-kitchensink-ear-web/pom.properties",
            "directory" => false,
            "file-size" => 146L
        }
    ]
}
```

You also have a **read-content** operation but since it returns a binary stream, this is not displayable from the CLI.

```
[domain@localhost:9990 /] /deployment=kitchensink.ear:read-content(path=META-INF/MANIFEST.MF)
{
    "outcome" => "success",
    "result" => {"uuid" => "b373d587-72ee-4b1e-a02a-71fbb0c85d32"},
    "response-headers" => {"attached-streams" => [{
        "uuid" => "b373d587-72ee-4b1e-a02a-71fbb0c85d32",
        "mime-type" => "text/plain"
    }]}
}
```

The management CLI however provides high level commands to display or save binary stream attachments:

```
[domain@localhost:9990 /] attachment display
--operation=/deployment=kitchensink.ear:read-content(path=META-INF/MANIFEST.MF)
ATTACHMENT d052340a-abb7-4a66-aa24-4eeeb6b256be:
Manifest-Version: 1.0
Archiver-Version: Plexus Archiver
Built-By: mjurc
Created-By: Apache Maven 3.3.9
Build-Jdk: 1.8.0_91
```

```
[domain@localhost:9990 /] attachment save
--operation=/deployment=kitchensink.ear:read-content(path=META-INF/MANIFEST.MF) --file=example
File saved to /home/mjurc/wildfly/build/target/wildfly-11.0.0.Alpha1-SNAPSHOT/example
```

XML Configuration File

When you deploy content, the domain controller adds two types of entries to the `domain.xml` configuration file, one showing global information about the deployment, and another for each relevant server group showing how it is used by that server group:

```
[...]
<deployments>
   <deployment name="test-application.war"
               runtime-name="test-application.war">
       <content sha1="dda9881fa7811b22f1424b4c5acccb13c71202bd"/>
   </deployment>
</deployments>
[...]
<server-groups>
   <server-group name="main-server-group" profile="default">
       [...]
       <deployments>
           <deployment name="test-application.war" runtime-name="test-application.war"/>
       </deployments>
   </server-group>
</server-groups>
[...]
```

(See domain/configuration/domain.xml)

5.6.2 Standalone Server

Deployments on a standalone server work in a similar way to those on managed domains. The main difference is that there are no server group associations.

Deployment Commands

The same CLI commands used for managed domains work for standalone servers when deploying and removing an application:

```
[standalone@localhost:9990 /] deploy ~/Desktop/test-application.war
'test-application.war' deployed successfully.

[standalone@localhost:9990 /] undeploy test-application.war
Successfully undeployed test-application.war.
```

Deploying Using the Deployment Scanner

Deployment content (for example, war, ear, jar, and sar files) can be placed in the standalone/deployments directory of the WildFly distribution, in order to be automatically deployed into the server runtime. For this to work the `deployment-scanner` subsystem must be present. The scanner periodically checks the contents of the deployments directory and reacts to changes by updating the server.

Users are encouraged to use the WildFly management APIs to upload and deploy deployment content instead of relying on the deployment scanner that periodically scans the directory, particularly if running production systems.

Deployment Scanner Modes

The WildFly filesystem deployment scanner operates in one of two different modes, depending on whether it will directly monitor the deployment content in order to decide to deploy or redeploy it.

Auto-deploy mode:

The scanner will directly monitor the deployment content, automatically deploying new content and redeploying content whose timestamp has changed. This is similiar to the behavior of previous AS releases, although there are differences:

- A change in any file in an exploded deployment triggers redeploy. Because EE 6+ applications do not require deployment descriptors,
  there is no attempt to monitor deployment descriptors and only redeploy when a deployment descriptor changes.
- The scanner will place marker files in this directory as an indication of the status of its attempts to deploy or undeploy content. These are detailed below.

Manual deploy mode:

The scanner will not attempt to directly monitor the deployment content and decide if or when the end user wishes the content to be deployed. Instead, the scanner relies on a system of marker files, with the user's addition or removal of a marker file serving as a sort of command telling the scanner to deploy, undeploy or redeploy content.

Auto-deploy mode and manual deploy mode can be independently configured for zipped deployment content and exploded deployment content. This is done via the "auto-deploy" attribute on the deployment-scanner element in the standalone.xml configuration file:

```
<deployment-scanner scan-interval="5000" relative-to="jboss.server.base.dir"
   path="deployments" auto-deploy-zipped="true" auto-deploy-exploded="false"/>
```

By default, auto-deploy of zipped content is enabled, and auto-deploy of exploded content is disabled. Manual deploy mode is strongly recommended for exploded content, as exploded content is inherently vulnerable to the scanner trying to auto-deploy partially copied content.

Marker Files

The marker files always have the same name as the deployment content to which they relate, but with an additional file suffix appended. For example, the marker file to indicate the example.war file should be deployed is named example.war.dodeploy. Different marker file suffixes have different meanings.

The relevant marker file types are:

File	Purpose
.dodeploy	Placed by the user to indicate that the given content should be deployed into the runtime (or redeployed if already deployed in the runtime.)
.skipdeploy	Disables auto-deploy of the content for as long as the file is present. Most useful for allowing updates to exploded content without having the scanner initiate redeploy in the middle of the update. Can be used with zipped content as well, although the scanner will detect in-progress changes to zipped content and wait until changes are complete.
.isdeploying	Placed by the deployment scanner service to indicate that it has noticed a .dodeploy file or new or updated auto-deploy mode content and is in the process of deploying the content. This marker file will be deleted when the deployment process completes.
.deployed	Placed by the deployment scanner service to indicate that the given content has been deployed into the runtime. If an end user deletes this file, the content will be undeployed.
.failed	Placed by the deployment scanner service to indicate that the given content failed to deploy into the runtime. The content of the file will include some information about the cause of the failure. Note that with auto-deploy mode, removing this file will make the deployment eligible for deployment again.
.isundeploying	Placed by the deployment scanner service to indicate that it has noticed a .deployed file has been deleted and the content is being undeployed. This marker file will be deleted when the undeployment process completes.
.undeployed	Placed by the deployment scanner service to indicate that the given content has been undeployed from the runtime. If an end user deletes this file, it has no impact.
.pending	Placed by the deployment scanner service to indicate that it has noticed the need to deploy content but has not yet instructed the server to deploy it. This file is created if the scanner detects that some auto-deploy content is still in the process of being copied or if there is some problem that prevents auto-deployment. The scanner will not instruct the server to deploy or undeploy any content (not just the directly affected content) as long as this condition holds.

Basic workflows:

All examples assume variable $JBOSS_HOME points to the root of the WildFly distribution.

A) Add new zipped content and deploy it:

1. cp target/example.war/ $JBOSS_HOME/standalone/deployments
2. (Manual mode only) touch $JBOSS_HOME/standalone/deployments/example.war.dodeploy

B) Add new unzipped content and deploy it:

1. cp -r target/example.war/ $JBOSS_HOME/standalone/deployments
2. (Manual mode only) touch $JBOSS_HOME/standalone/deployments/example.war.dodeploy

C) Undeploy currently deployed content:

1. rm $JBOSS_HOME/standalone/deployments/example.war.deployed

D) Auto-deploy mode only: Undeploy currently deployed content:

1. rm $JBOSS_HOME/standalone/deployments/example.war

E) Replace currently deployed zipped content with a new version and deploy it:

1. cp target/example.war/ $JBOSS_HOME/standalone/deployments
2. (Manual mode only) touch $JBOSS_HOME/standalone/deployments/example.war.dodeploy

F) Manual mode only: Replace currently deployed unzipped content with a new version and deploy it:

1. rm $JBOSS_HOME/standalone/deployments/example.war.deployed
2. wait for $JBOSS_HOME/standalone/deployments/example.war.undeployed file to appear
3. cp -r target/example.war/ $JBOSS_HOME/standalone/deployments
4. touch $JBOSS_HOME/standalone/deployments/example.war.dodeploy

G) Auto-deploy mode only: Replace currently deployed unzipped content with a new version and deploy it:

1. touch $JBOSS_HOME/standalone/deployments/example.war.skipdeploy
2. cp -r target/example.war/ $JBOSS_HOME/standalone/deployments
3. rm $JBOSS_HOME/standalone/deployments/example.war.skipdeploy

H) Manual mode only: Live replace portions of currently deployed unzipped content without redeploying:

1. cp -r target/example.war/foo.html $JBOSS_HOME/standalone/deployments/example.war

I) Auto-deploy mode only: Live replace portions of currently deployed unzipped content without redeploying:

1. touch $JBOSS_HOME/standalone/deployments/example.war.skipdeploy
2. cp -r target/example.war/foo.html $JBOSS_HOME/standalone/deployments/example.war

J) Manual or auto-deploy mode: Redeploy currently deployed content (i.e. bounce it with no content change):

1. touch $JBOSS_HOME/standalone/deployments/example.war.dodeploy

K) Auto-deploy mode only: Redeploy currently deployed content (i.e. bounce it with no content change):

1. touch $JBOSS_HOME/standalone/deployments/example.war

The above examples use Unix shell commands. Windows equivalents are:

cp src dest --> xcopy /y src dest
cp -r src dest --> xcopy /e /s /y src dest
rm afile --> del afile
touch afile --> echo>> afile

Note that the behavior of 'touch' and 'echo' are different but the differences are not relevant to the usages in the examples above.

5.6.3 Managed and Unmanaged Deployments

WildFly supports two mechanisms for dealing with deployment content – managed and unmanaged deployments.

With a managed deployment the server takes the deployment content and copies it into an internal content repository and thereafter uses that copy of the content, not the original user-provided content. The server is thereafter responsible for the content it uses.

With an unmanaged deployment the user provides the local filesystem path of deployment content, and the server directly uses that content. However the user is responsible for ensuring that content, e.g. for making sure that no changes are made to it that will negatively impact the functioning of the deployed application.

To help you differentiate managed from unmanaged deployments the deployment model has a runtime boolean attribute 'managed'.

Managed deployments have a number of benefits over unmanaged:

- They can be manipulated by remote management clients, not requiring access to the server filesystem.
- In a managed domain, WildFly/EAP will take responsibility for replicating a copy of the deployment to all hosts/servers in the domain where it is needed. With an unmanaged deployment, it is the user's responsibility to have the deployment available on the local filesystem on all relevant hosts, at a consistent path.
- The deployment content actually used is stored on the filesystem in the internal content repository, which should help shelter it from unintended changes.

All of the previous examples above illustrate using managed deployments, except for any discussion of deployment scanner handling of exploded deployments. In WildFly 10 and earlier exploded deployments are always unmanaged, this is no longer the case since WildFly 11.

Content Repository

For a managed deployment, the actual file the server uses when creating runtime services is not the file provided to the CLI `deploy` command or to the web console. It is a copy of that file stored in an internal content repository. The repository is located in the `domain/data/content` directory for a managed domain, or in `standalone/data/content` for a standalone server. Actual binaries are stored in a subdirectory:

```
ls domain/data/content/
  |---/47
  |-----95cc29338b5049e238941231b36b3946952991
  |---/dd
  |-----a9881fa7811b22f1424b4c5acccb13c71202bd
```

The location of the content repository and its internal structure is subject to change at any time and should not be relied upon by end users.

The description of a managed deployment in the domain or standalone configuration file includes an attribute recording the SHA1 hash of the deployment content:

```
<deployments>
   <deployment name="test-application.war"
               runtime-name="test-application.war">
       <content sha1="dda9881fa7811b22f1424b4c5acccb13c71202bd"/>
   </deployment>
</deployments>
```

The WildFly process calculates and records that hash when the user invokes a management operation (e.g. CLI `deploy` command or using the console) providing deployment content. The user is not expected to calculate the hash.

The sha1 attribute in the content element tells the WildFly process where to find the deployment content in its internal content repository.

In a domain each host will have a copy of the content needed by its servers in its own local content repository. The WildFly domain controller and slave host controller processes take responsibility for ensuring each host has the needed content.

Unmanaged Deployments

An unmanaged deployment is one where the server directly deploys the content at a path you specify instead of making an internal copy and then deploying the copy.

Initially deploying an unmanaged deployment is much like deploying a managed one, except you tell WildFly that you do not want the deployment to be managed:

```
[standalone@localhost:9990 /] deploy ~/Desktop/test-application.war --unmanaged
'test-application.war' deployed successfully.
```

When you do this, instead of the server making a copy of the content at `/Desktop/test-application.war`, calculating the hash of the content, storing the hash in the configuration file and then installing the copy into the runtime, instead it will convert `/Desktop/test-application.war` to an absolute path, store the path in the configuration file, and then install the original content in the runtime.

You can also use unmanaged deployments in a domain:

```
[domain@localhost:9990 /] deploy /home/example/Desktop/test-application.war
--server-group=main-server-group --unmanaged
'test-application.war' deployed successfully.
```

However, before you run this command you must ensure that a copy of the content is present on all machines that have servers in the target server groups, all at the same filesystem path. The domain will not copy the file for you.

Undeploy is no different from a managed undeploy:

```
[standalone@localhost:9990 /] undeploy test-application.war
Successfully undeployed test-application.war.
```

Doing a replacement of the deployment with a new version is a bit different, the server is using the file you want to replace. You should undeploy the deployment, replace the content, and then deploy again. Or you can stop the server, replace the deployment and deploy again.

5.6.4 Deployment overlays

Deployment overlays are our way of 'overlaying' content into an existing deployment, without physically modifying the contents of the deployment archive. Possible use cases include swapping out deployment descriptors, modifying static web resources to change the branding of an application, or even replacing jar libraries with different versions.

Deployment overlays have a different lifecycle to a deployment. In order to use a deployment overlay, you first create the overlay, using the CLI or the management API. You then add files to the overlay, specifying the deployment paths you want them to overlay. Once you have created the overlay you then have to link it to a deployment name (which is done slightly differently depending on if you are in standalone or domain mode). Once you have created the link any deployment that matches the specified deployment name will have the overlay applied.

When you modify or create an overlay it will not affect existing deployments, they must be redeployed in order to take effect

Creating a deployment overlay

To create a deployment overlay the CLI provides a high level command to do all the steps specified above in one go. An example command is given below for both standalone and domain mode:

```
deployment-overlay add --name=myOverlay
--content=/WEB-INF/web.xml=/myFiles/myWeb.xml,/WEB-INF/ejb-jar.xml=/myFiles/myEjbJar.xml
--deployments=test.war,*-admin.war --redeploy-affected
```

```
deployment-overlay add --name=myOverlay
--content=/WEB-INF/web.xml=/myFiles/myWeb.xml,/WEB-INF/ejb-jar.xml=/myFiles/myEjbJar.xml
--deployments=test.war,*-admin.war --server-groups=main-server-group --redeploy-affected
```

5.7 Subsystem configuration

The following chapters will focus on the high level management use cases that are available through the CLI and the web interface. For a detailed description of each subsystem configuration property, please consult the respective component reference.

Schema Location

The configuration schemas can found in `$JBOSS_HOME/docs/schema`.

5.7.1 EE Subsystem Configuration

Overview

The EE subsystem provides common functionality in the Java EE platform, such as the EE Concurrency Utilities (JSR 236) and `@Resource` injection. The subsystem is also responsible for managing the lifecycle of Java EE application's deployments, that is, `.ear` files.

The EE subsystem configuration may be used to:

- customise the deployment of Java EE applications
- create EE Concurrency Utilities instances
- define the default bindings

The subsystem name is ee and this document covers EE subsystem version `2.0`, which XML namespace within WildFly XML configurations is `urn:jboss:domain:ee:2.0`. The path for the subsystem's XML schema, within WildFly's distribution, is `docs/schema/jboss-as-ee_2_0.xsd`.

Subsystem XML configuration example with all elements and attributes specified:

```
<subsystem xmlns="urn:jboss:domain:ee:2.0" >
    <global-modules>
        <module name="org.jboss.logging"
                slot="main"/>
        <module name="org.apache.log4j"
                annotations="true"
                meta-inf="true"
                services="false" />
    </global-modules>
    <ear-subdeployments-isolated>true</ear-subdeployments-isolated>
    <spec-descriptor-property-replacement>false</spec-descriptor-property-replacement>
    <jboss-descriptor-property-replacement>false</jboss-descriptor-property-replacement>
    <annotation-property-replacement>false</annotation-property-replacement>
    <concurrent>
        <context-services>
            <context-service
                    name="default"
                    jndi-name="java:jboss/ee/concurrency/context/default"
                    use-transaction-setup-provider="true" />
        </context-services>
        <managed-thread-factories>
            <managed-thread-factory
                    name="default"
                    jndi-name="java:jboss/ee/concurrency/factory/default"
                    context-service="default"
                    priority="1" />
        </managed-thread-factories>
        <managed-executor-services>
            <managed-executor-service
                    name="default"
                    jndi-name="java:jboss/ee/concurrency/executor/default"
```

```
                    context-service="default"
                    thread-factory="default"
                    hung-task-threshold="60000"
                    core-threads="5"
                    max-threads="25"
                    keepalive-time="5000"
                    queue-length="1000000"
                    reject-policy="RETRY_ABORT" />
        </managed-executor-services>
        <managed-scheduled-executor-services>
            <managed-scheduled-executor-service
                    name="default"
                    jndi-name="java:jboss/ee/concurrency/scheduler/default"
                    context-service="default"
                    thread-factory="default"
                    hung-task-threshold="60000"
                    core-threads="5"
                    keepalive-time="5000"
                    reject-policy="RETRY_ABORT" />
        </managed-scheduled-executor-services>
    </concurrent>
    <default-bindings
            context-service="java:jboss/ee/concurrency/context/default"
            datasource="java:jboss/datasources/ExampleDS"
            jms-connection-factory="java:jboss/DefaultJMSConnectionFactory"
            managed-executor-service="java:jboss/ee/concurrency/executor/default"
            managed-scheduled-executor-service="java:jboss/ee/concurrency/scheduler/default"
            managed-thread-factory="java:jboss/ee/concurrency/factory/default" />
</subsystem>
```

Java EE Application Deployment

The EE subsystem configuration allows the customisation of the deployment behaviour for Java EE Applications.

Global Modules

Global modules is a set of JBoss Modules that will be added as dependencies to the JBoss Module of every Java EE deployment. Such dependencies allows Java EE deployments to see the classes exported by the global modules.

Each global module is defined through the `module` resource, an example of its XML configuration:

```
<global-modules>
    <module name="org.jboss.logging" slot="main"/>
    <module name="org.apache.log4j" annotations="true" meta-inf="true" services="false" />
  </global-modules>
```

The only mandatory attribute is the JBoss Module `name`, the `slot` attribute defaults to `main`, and both define the JBoss Module ID to reference.

The optional `annotations` attribute, which defaults to `false`, indicates if a pre-computed annotation index should be imported from META-INF/jandex.idx

The optional `services` attribute indicates if any services exposed in META-INF/services should be made available to the deployments class loader, and defaults to `false`.

The optional `meta-inf` attribute, which defaults to `true`, indicates if the Module's `META-INF` path should be available to the deployment's class loader.

EAR Subdeployments Isolation

A flag indicating whether each of the subdeployments within a `.ear` can access classes belonging to another subdeployment within the same `.ear`. The default value is `false`, which allows the subdeployments to see classes belonging to other subdeployments within the `.ear`.

```
<ear-subdeployments-isolated>true</ear-subdeployments-isolated>
```

For example:

```
myapp.ear
|
|--- web.war
|
|--- ejb1.jar
|
|--- ejb2.jar
```

If the `ear-subdeployments-isolated` is set to false, then the classes in `web.war` can access classes belonging to `ejb1.jar` and `ejb2.jar`. Similarly, classes from `ejb1.jar` can access classes from `ejb2.jar` (and vice-versa).

This flag has no effect on the isolated classloader of the `.war` file(s), i.e. irrespective of whether this flag is set to `true` or `false`, the `.war` within a `.ear` will have a isolated classloader, and other subdeployments within that `.ear` will not be able to access classes from that `.war`. This is as per spec.

Property Replacement

The EE subsystem configuration includes flags to configure whether system property replacement will be done on XML descriptors and Java Annotations included in Java EE deployments.

System properties etc are resolved in the security context of the application server itself, not the deployment that contains the file. This means that if you are running with a security manager and enable this property, a deployment can potentially access system properties or environment entries that the security manager would have otherwise prevented.

Spec Descriptor Property Replacement

Flag indicating whether system property replacement will be performed on standard Java EE XML descriptors. If not configured this defaults to `true`, however it is set to `false` in the standard configuration files shipped with WildFly.

```
<spec-descriptor-property-replacement>false</spec-descriptor-property-replacement>
```

JBoss Descriptor Property Replacement

Flag indicating whether system property replacement will be performed on WildFly proprietary XML descriptors, such as `jboss-app.xml`. This defaults to `true`.

```
<jboss-descriptor-property-replacement>false</jboss-descriptor-property-replacement>
```

Annotation Property Replacement

Flag indicating whether system property replacement will be performed on Java annotations. The default value is `false`.

```
<annotation-property-replacement>false</annotation-property-replacement>
```

EE Concurrency Utilities

EE Concurrency Utilities (JSR 236) were introduced with Java EE 7, to ease the task of writing multithreaded Java EE applications. Instances of these utilities are managed by WildFly, and the related configuration provided by the EE subsystem.

Context Services

The Context Service is a concurrency utility which creates contextual proxies from existent objects. WildFly Context Services are also used to propagate the context from a Java EE application invocation thread, to the threads internally used by the other EE Concurrency Utilities. Context Service instances may be created using the subsystem XML configuration:

```
<context-services>
    <context-service
 name="default"
 jndi-name="java:jboss/ee/concurrency/context/default"
 use-transaction-setup-provider="true" />
  </context-services>
```

The `name` attribute is mandatory, and it's value should be a unique name within all Context Services.

The `jndi-name` attribute is also mandatory, and defines where in the JNDI the Context Service should be placed.

The optional `use-trasaction-setup-provider` attribute indicates if the contextual proxies built by the Context Service should suspend transactions in context, when invoking the proxy objects, and its value defaults to true.

Management clients, such as the WildFly CLI, may also be used to configure Context Service instances. An example to `add` and `remove` one named `other`:

```
/subsystem=ee/context-service=other:add(jndi-name=java\:jboss\/ee\/concurrency\/other)
/subsystem=ee/context-service=other:remove
```

Managed Thread Factories

The Managed Thread Factory allows Java EE applications to create new threads. WildFly Managed Thread Factory instances may also, optionally, use a Context Service instance to propagate the Java EE application thread's context to the new threads. Instance creation is done through the EE subsystem, by editing the subsystem XML configuration:

```
<managed-thread-factories>
    <managed-thread-factory
 name="default"
 jndi-name="java:jboss/ee/concurrency/factory/default"
 context-service="default"
 priority="1" />
  </managed-thread-factories>
```

The `name` attribute is mandatory, and it's value should be a unique name within all Managed Thread Factories.

The `jndi-name` attribute is also mandatory, and defines where in the JNDI the Managed Thread Factory should be placed.

The optional `context-service` references an existent Context Service by its `name`. If specified then thread created by the factory will propagate the invocation context, present when creating the thread.

The optional `priority` indicates the priority for new threads created by the factory, and defaults to `5`.

Management clients, such as the WildFly CLI, may also be used to configure Managed Thread Factory instances. An example to `add` and `remove` one named `other`:

```
/subsystem=ee/managed-thread-factory=other:add(jndi-name=java\:jboss\/ee\/factory\/other)
/subsystem=ee/managed-thread-factory=other:remove
```

Managed Executor Services

The Managed Executor Service is the Java EE adaptation of Java SE Executor Service, providing to Java EE applications the functionality of asynchronous task execution. WildFly is responsible to manage the lifecycle of Managed Executor Service instances, which are specified through the EE subsystem XML configuration:

```
<managed-executor-services>
    <managed-executor-service
 name="default"
 jndi-name="java:jboss/ee/concurrency/executor/default"
 context-service="default"
 thread-factory="default"
 hung-task-threshold="60000"
 core-threads="5"
 max-threads="25"
 keepalive-time="5000"
 queue-length="1000000"
 reject-policy="RETRY_ABORT" />
  </managed-executor-services>
```

The `name` attribute is mandatory, and it's value should be a unique name within all Managed Executor Services.

The `jndi-name` attribute is also mandatory, and defines where in the JNDI the Managed Executor Service should be placed.

The optional `context-service` references an existent Context Service by its `name`. If specified then the referenced Context Service will capture the invocation context present when submitting a task to the executor, which will then be used when executing the task.

The optional `thread-factory` references an existent Managed Thread Factory by its `name`, to handle the creation of internal threads. If not specified then a Managed Thread Factory with default configuration will be created and used internally.

The mandatory `core-threads` provides the number of threads to keep in the executor's pool, even if they are idle. A value of `0` means there is no limit.

The optional `queue-length` indicates the number of tasks that can be stored in the input queue. The default value is `0`, which means the queue capacity is unlimited.

The executor’s task queue is based on the values of the attributes `core-threads` and `queue-length`:

- If `queue-length` is `0`, or `queue-length` is `Integer.MAX_VALUE (2147483647)` and `core-threads` is `0`, direct handoff queuing strategy will be used and a synchronous queue will be created.
- If `queue-length` is `Integer.MAX_VALUE` but `core-threads` is not `0`, an unbounded queue will be used.
- For any other valid value for `queue-length`, a bounded queue wil be created.

The optional `hung-task-threshold` defines a threshold value, in milliseconds, to hung a possibly blocked task. A value of `0` will never hung a task, and is the default.

The optional `long-running-tasks` is a hint to optimize the execution of long running tasks, and defaults to `false`.

The optional `max-threads` defines the the maximum number of threads used by the executor, which defaults to Integer.MAX_VALUE (2147483647).

The optional `keepalive-time` defines the time, in milliseconds, that an internal thread may be idle. The attribute default value is `60000`.

The optional reject-policy defines the policy to use when a task is rejected by the executor. The attribute value may be the default `ABORT`, which means an exception should be thrown, or `RETRY_ABORT`, which means the executor will try to submit it once more, before throwing an exception.

Management clients, such as the WildFly CLI, may also be used to configure Managed Executor Service instances. An example to `add` and `remove` one named `other`:

```
/subsystem=ee/managed-executor-service=other:add(jndi-name=java\:jboss\/ee\/executor\/other,
core-threads=2)
/subsystem=ee/managed-executor-service=other:remove
```

Managed Scheduled Executor Services

The Managed Scheduled Executor Service is the Java EE adaptation of Java SE Scheduled Executor Service, providing to Java EE applications the functionality of scheduling task execution. WildFly is responsible to manage the lifecycle of Managed Scheduled Executor Service instances, which are specified through the EE subsystem XML configuration:

```
<managed-scheduled-executor-services>
    <managed-scheduled-executor-service
 name="default"
 jndi-name="java:jboss/ee/concurrency/scheduler/default"
 context-service="default"
 thread-factory="default"
 hung-task-threshold="60000"
 core-threads="5"
 keepalive-time="5000"
 reject-policy="RETRY_ABORT" />
  </managed-scheduled-executor-services>
```

The `name` attribute is mandatory, and it's value should be a unique name within all Managed Scheduled Executor Services.

The `jndi-name` attribute is also mandatory, and defines where in the JNDI the Managed Scheduled Executor Service should be placed.

The optional `context-service` references an existent Context Service by its `name`. If specified then the referenced Context Service will capture the invocation context present when submitting a task to the executor, which will then be used when executing the task.

The optional `thread-factory` references an existent Managed Thread Factory by its `name`, to handle the creation of internal threads. If not specified then a Managed Thread Factory with default configuration will be created and used internally.

The mandatory `core-threads` provides the number of threads to keep in the executor's pool, even if they are idle. A value of `0` means there is no limit.

The optional `hung-task-threshold` defines a threshold value, in milliseconds, to hung a possibly blocked task. A value of `0` will never hung a task, and is the default.

The optional `long-running-tasks` is a hint to optimize the execution of long running tasks, and defaults to `false`.

The optional `keepalive-time` defines the time, in milliseconds, that an internal thread may be idle. The attribute default value is `60000`.

The optional reject-policy defines the policy to use when a task is rejected by the executor. The attribute value may be the default `ABORT`, which means an exception should be thrown, or`RETRY_ABORT`, which means the executor will try to submit it once more, before throwing an exception.

Management clients, such as the WildFly CLI, may also be used to configure Managed Scheduled Executor Service instances. An example to `add` and `remove` one named `other`:

```
/subsystem=ee/managed-scheduled-executor-service=other:add(jndi-name=java\:jboss\/ee\/scheduler\/o
core-threads=2)
/subsystem=ee/managed-scheduled-executor-service=other:remove
```

Default EE Bindings

The Java EE Specification mandates the existence of a default instance for each of the following resources:

- Context Service
- Datasource
- JMS Connection Factory
- Managed Executor Service
- Managed Scheduled Executor Service
- Managed Thread Factory

The EE subsystem looks up the default instances from JNDI, using the names in the default bindings configuration, before placing those in the standard JNDI names, such as `java:comp/DefaultManagedExecutorService`:

```
<default-bindings
 context-service="java:jboss/ee/concurrency/context/default"
 datasource="java:jboss/datasources/ExampleDS"
 jms-connection-factory="java:jboss/DefaultJMSConnectionFactory"
 managed-executor-service="java:jboss/ee/concurrency/executor/default"
 managed-scheduled-executor-service="java:jboss/ee/concurrency/scheduler/default"
 managed-thread-factory="java:jboss/ee/concurrency/factory/default" />
```

The default bindings are optional, if the jndi name for a default binding is not configured then the related resource will not be available to Java EE applications.

5.7.2 Naming

Overview

The Naming subsystem provides the JNDI implementation on WildFly, and its configuration allows to:

- bind entries in global JNDI namespaces
- turn off/on the remote JNDI interface

The subsystem name is naming and this document covers Naming subsystem version `2.0`, which XML namespace within WildFly XML configurations is `urn:jboss:domain:naming:2.0`. The path for the subsystem's XML schema, within WildFly's distribution, is `docs/schema/jboss-as-naming_2_0.xsd`.

Subsystem XML configuration example with all elements and attributes specified:

```
<subsystem xmlns="urn:jboss:domain:naming:2.0">
    <bindings>
        <simple name="java:global/a" value="100" type="int" />
        <simple name="java:global/jboss.org/docs/url" value="https://docs.jboss.org"
type="java.net.URL" />
        <object-factory name="java:global/foo/bar/factory" module="org.foo.bar"
class="org.foo.bar.ObjectFactory" />
        <external-context name="java:global/federation/ldap/example"
class="javax.naming.directory.InitialDirContext" cache="true">
            <environment>
                <property name="java.naming.factory.initial"
value="com.sun.jndi.ldap.LdapCtxFactory" />
                <property name="java.naming.provider.url" value="ldap://ldap.example.com:389" />
                <property name="java.naming.security.authentication" value="simple" />
                <property name="java.naming.security.principal" value="uid=admin,ou=system" />
                <property name="java.naming.security.credentials" value="secret" />
            </environment>
        </external-context>
        <lookup name="java:global/c" lookup="java:global/b" />
    </bindings>
    <remote-naming/>
</subsystem>
```

Global Bindings Configuration

The Naming subsystem configuration allows binding entries into the following global JNDI namespaces:

- `java:global`
- `java:jboss`
- `java:`

If WildFly is to be used as a Java EE application server, then it's recommended to opt for `java:global`, since it is a standard (i.e. portable) namespace.

Four different types of bindings are supported:

- Simple
- Object Factory
- External Context
- Lookup

In the subsystem's XML configuration, global bindings are configured through the `<bindings />` XML element, as an example:

```
<bindings>
    <simple name="java:global/a" value="100" type="int" />
    <object-factory name="java:global/foo/bar/factory" module="org.foo.bar"
class="org.foo.bar.ObjectFactory" />
    <external-context name="java:global/federation/ldap/example"
class="javax.naming.directory.InitialDirContext" cache="true">
        <environment>
            <property name="java.naming.factory.initial"
value="com.sun.jndi.ldap.LdapCtxFactory" />
            <property name="java.naming.provider.url" value="ldap://ldap.example.com:389" />
            <property name="java.naming.security.authentication" value="simple" />
            <property name="java.naming.security.principal" value="uid=admin,ou=system" />
            <property name="java.naming.security.credentials" value="secret" />
        </environment>
    </external-context>
    <lookup name="java:global/c" lookup="java:global/b" />
</bindings>
```

Simple Bindings

A simple binding is a primitive or java.net.URL entry, and it is defined through the `simple` XML element. An example of its XML configuration:

```
<simple name="java:global/a" value="100" type="int" />
```

The `name` attribute is mandatory and specifies the target JNDI name for the entry.

The `value` attribute is mandatory and defines the entry's value.

The optional `type` attribute, which defaults to `java.lang.String`, specifies the type of the entry's value. Besides java.lang.String, allowed types are all the primitive types and their corresponding object wrapper classes, such as int or java.lang.Integer, and java.net.URL.

Management clients, such as the WildFly CLI, may be used to configure simple bindings. An example to `add` and `remove` the one in the XML example above:

```
/subsystem=naming/binding=java\:global\/a:add(binding-type=simple, type=int, value=100)
/subsystem=naming/binding=java\:global\/a:remove
```

Object Factories

The Naming subsystem configuration allows the binding of `javax.naming.spi.ObjectFactory` entries, through the `object-factory` XML element, for instance:

```
<object-factory name="java:global/foo/bar/factory" module="org.foo.bar"
class="org.foo.bar.ObjectFactory">
    <environment>
        <property name="p1" value="v1" />
        <property name="p2" value="v2" />
    </environment>
</object-factory>
```

The `name` attribute is mandatory and specifies the target JNDI name for the entry.

The `class` attribute is mandatory and defines the object factory's Java type.

The `module` attribute is mandatory and specifies the JBoss Module ID where the object factory Java class may be loaded from.

The optional `environment` child element may be used to provide a custom environment to the object factory.

Management clients, such as the WildFly CLI, may be used to configure object factory bindings. An example to `add` and `remove` the one in the XML example above:

```
/subsystem=naming/binding=java\:global\/foo\/bar\/factory:add(binding-type=object-factory,
module=org.foo.bar, class=org.foo.bar.ObjectFactory, environment=[p1=v1, p2=v2])
/subsystem=naming/binding=java\:global\/foo\/bar\/factory:remove
```

External Context Federation

Federation of external JNDI contexts, such as a LDAP context, are achieved by adding External Context bindings to the global bindings configuration, through the `external-context` XML element. An example of its XML configuration:

```
<external-context name="java:global/federation/ldap/example"
class="javax.naming.directory.InitialDirContext" cache="true">
    <environment>
        <property name="java.naming.factory.initial" value="com.sun.jndi.ldap.LdapCtxFactory" />
        <property name="java.naming.provider.url" value="ldap://ldap.example.com:389" />
        <property name="java.naming.security.authentication" value="simple" />
        <property name="java.naming.security.principal" value="uid=admin,ou=system" />
        <property name="java.naming.security.credentials" value="secret" />
    </environment>
</external-context>
```

The `name` attribute is mandatory and specifies the target JNDI name for the entry.

The `class` attribute is mandatory and indicates the Java initial naming context type used to create the federated context. Note that such type must have a constructor with a single environment map argument.

The optional `module` attribute specifies the JBoss Module ID where any classes required by the external JNDI context may be loaded from.

The optional `cache` attribute, which value defaults to `false`, indicates if the external context instance should be cached.

The optional `environment` child element may be used to provide the custom environment needed to lookup the external context.

Management clients, such as the WildFly CLI, may be used to configure external context bindings. An example to `add` and `remove` the one in the XML example above:

```
/subsystem=naming/binding=java\:global\/federation\/ldap\/example:add(binding-type=external-contex
cache=true, class=javax.naming.directory.InitialDirContext,
environment=[java.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory,
java.naming.provider.url=ldap\:\/\/ldap.example.com\:389,
java.naming.security.authentication=simple,
java.naming.security.principal=uid\=admin\,ou\=system, java.naming.security.credentials=
secret])

/subsystem=naming/binding=java\:global\/federation\/ldap\/example:remove
```

Some JNDI providers may fail when their resources are looked up if they do not implement properly the lookup(Name) method. Their errors would look like:

```
11:31:49,047 ERROR org.jboss.resource.adapter.jms.inflow.JmsActivation (default-threads
-1)  javax.naming.InvalidNameException: Only support CompoundName names
    at com.tibco.tibjms.naming.TibjmsContext.lookup(TibjmsContext.java:504)
    at javax.naming.InitialContext.lookup(InitialContext.java:421)
```

To work around their shortcomings, the `org.jboss.as.naming.lookup.by.string` property can be specified in the external-context's environment to use instead the lookup(String) method (with a performance degradation):

```
<property name="org.jboss.as.naming.lookup.by.string" value="true"/>
```

Binding Aliases

The Naming subsystem configuration allows the binding of existent entries into additional names, i.e. aliases. Binding aliases are specified through the `lookup` XML element. An example of its XML configuration:

```
<lookup name="java:global/c" lookup="java:global/b" />
```

The `name` attribute is mandatory and specifies the target JNDI name for the entry.

The `lookup` attribute is mandatory and indicates the source JNDI name. It can chain lookups on external contexts. For example, having an external context bounded to *java:global/federation/ldap/example*, searching can be done there by setting `lookup` attribute to *java:global/federation/ldap/example/subfolder*.

Management clients, such as the WildFly CLI, may be used to configure binding aliases. An example to `add` and `remove` the one in the XML example above:

```
/subsystem=naming/binding=java\:global\/c:add(binding-type=lookup, lookup=java\:global\/b)
/subsystem=naming/binding=java\:global\/c:remove
```

Remote JNDI Configuration

The Naming subsystem configuration may be used to (de)activate the remote JNDI interface, which allows clients to lookup entries present in a remote WildFly instance.

 Only entries within the `java:jboss/exported` context are accessible over remote JNDI.

In the subsystem's XML configuration, remote JNDI access bindings are configured through the `<remote-naming />` XML element:

```
<remote-naming />
```

Management clients, such as the WildFly CLI, may be used to add/remove the remote JNDI interface. An example to `add` and `remove` the one in the XML example above:

```
/subsystem=naming/service=remote-naming:add
/subsystem=naming/service=remote-naming:remove
```

5.7.3 Data sources

Datasources are configured through the *datasource* subsystem. Declaring a new datasource consists of two separate steps: You would need to provide a JDBC driver and define a datasource that references the driver you installed.

JDBC Driver Installation

The recommended way to install a JDBC driver into WildFly 8 is to deploy it as a regular JAR deployment. The reason for this is that when you run WildFly in domain mode, deployments are automatically propagated to all servers to which the deployment applies; thus distribution of the driver JAR is one less thing for you to worry about!

Any JDBC 4-compliant driver will automatically be recognized and installed into the system by name and version. A JDBC JAR is identified using the Java service provider mechanism. Such JARs will contain a text a file named `META-INF/services/java.sql.Driver`, which contains the name of the class(es) of the Drivers which exist in that JAR. If your JDBC driver JAR is not JDBC 4-compliant, it can be made deployable in one of a few ways.

Modify the JAR

The most straightforward solution is to simply modify the JAR and add the missing file. You can do this from your command shell by:

1. Change to, or create, an empty temporary directory.
2. Create a `META-INF` subdirectory.
3. Create a `META-INF/services` subdirectory.
4. Create a `META-INF/services/java.sql.Driver` file which contains one line - the fully-qualified class name of the JDBC driver.
5. Use the `jar` command-line tool to update the JAR like this:

```
jar \-uf jdbc-driver.jar META-INF/services/java.sql.Driver
```

For a detailed explanation how to deploy JDBC 4 compliant driver jar, please refer to the chapter "Application Deployment".

Datasource Definitions

The datasource itself is defined within the subsystem *datasources*:

```
<subsystem xmlns="urn:jboss:domain:datasources:4.0">
    <datasources>
        <datasource jndi-name="java:jboss/datasources/ExampleDS" pool-name="ExampleDS">
            <connection-url>jdbc:h2:mem:test;DB_CLOSE_DELAY=-1</connection-url>
            <driver>h2</driver>
            <pool>
                <min-pool-size>10</min-pool-size>
                <max-pool-size>20</max-pool-size>
                <prefill>true</prefill>
            </pool>
            <security>
                <user-name>sa</user-name>
                <password>sa</password>
            </security>
        </datasource>
        <xa-datasource jndi-name="java:jboss/datasources/ExampleXADS" pool-name="ExampleXADS">
           <driver>h2</driver>
           <xa-datasource-property name="URL">jdbc:h2:mem:test</xa-datasource-property>
           <xa-pool>
                <min-pool-size>10</min-pool-size>
                <max-pool-size>20</max-pool-size>
                <prefill>true</prefill>
           </xa-pool>
           <security>
                <user-name>sa</user-name>
                <password>sa</password>
           </security>
        </xa-datasource>
        <drivers>
            <driver name="h2" module="com.h2database.h2">
                <xa-datasource-class>org.h2.jdbcx.JdbcDataSource</xa-datasource-class>
            </driver>
        </drivers>
  </datasources>

</subsystem>
```

(See `standalone/configuration/standalone.xml`)

As you can see the datasource references a driver by it's logical name.

You can easily query the same information through the CLI:

```
[standalone@localhost:9990 /] /subsystem=datasources:read-resource(recursive=true)
{
    "outcome" => "success",
    "result" => {
        "data-source" => {"H2DS" => {
            "connection-url" => "jdbc:h2:mem:test;DB_CLOSE_DELAY=-1",
            "jndi-name" => "java:/H2DS",
            "driver-name" => "h2",
            "pool-name" => "H2DS",
            "use-java-context" => true,
            "enabled" => true,
            "jta" => true,
            "pool-prefill" => true,
            "pool-use-strict-min" => false,
            "user-name" => "sa",
            "password" => "sa",
            "flush-strategy" => "FailingConnectionOnly",
            "background-validation" => false,
            "use-fast-fail" => false,
            "validate-on-match" => false,
            "use-ccm" => true
        }},
        "xa-data-source" => undefined,
        "jdbc-driver" => {"h2" => {
            "driver-name" => "h2",
            "driver-module-name" => "com.h2database.h2",
            "driver-xa-datasource-class-name" => "org.h2.jdbcx.JdbcDataSource"
        }}
    }
}


[standalone@localhost:9990 /] /subsystem=datasources:installed-drivers-list
{
    "outcome" => "success",
    "result" => [{
        "driver-name" => "h2",
        "deployment-name" => undefined,
        "driver-module-name" => "com.h2database.h2",
        "module-slot" => "main",
        "driver-xa-datasource-class-name" => "org.h2.jdbcx.JdbcDataSource",
        "driver-class-name" => "org.h2.Driver",
        "driver-major-version" => 1,
        "driver-minor-version" => 3,
        "jdbc-compliant" => true
    }]
}
```

Using the web console or the CLI greatly simplifies the deployment of JDBC drivers and the creation of datasources.

The CLI offers a set of commands to create and modify datasources:

```
[standalone@localhost:9990 /] data-source --help

SYNOPSIS
  data-source --help [--properties | --commands] |
              (--name=<resource_id> (--<property>=<value>)*) |
              (<command> --name=<resource_id> (--<parameter>=<value>)*)
              [--headers={<operation_header> (;<operation_header>)*}]
DESCRIPTION
  The command is used to manage resources of type /subsystem=datasources/data-source.
[...]


[standalone@localhost:9990 /] xa-data-source --help

SYNOPSIS
  xa-data-source --help [--properties | --commands] |
                 (--name=<resource_id> (--<property>=<value>)*) |
                 (<command> --name=<resource_id> (--<parameter>=<value>)*)
                 [--headers={<operation_header> (;<operation_header>)*}]

DESCRIPTION
  The command is used to manage resources of type /subsystem=datasources/xa-data-source.

RESOURCE DESCRIPTION
  A JDBC XA data-source configuration

[...]
```

Using security domains

Information can be found at https://community.jboss.org/wiki/JBossAS7SecurityDomainModel

Component Reference

The datasource subsystem is provided by the IronJacamar project. For a detailed description of the available configuration properties, please consult the project documentation.

- IronJacamar homepage: http://ironjacamar.org/
- Project Documentation: http://ironjacamar.org/documentation.html
- Schema description: http://www.ironjacamar.org/doc/userguide/1.1/en-US/html_single/index.html#deployingds_descriptor

5.7.4 Logging

Overview

The overall server logging configuration is represented by the logging subsystem. It consists of four notable parts: `handler` configurations, `logger`, the `root logger` declarations (aka log categories) and logging profiles. Each logger does reference a handler (or set of handlers). Each handler declares the log format and output:

```
<subsystem xmlns="urn:jboss:domain:logging:3.0">
   <console-handler name="CONSOLE" autoflush="true">
       <level name="DEBUG"/>
       <formatter>
           <named-formatter name="COLOR-PATTERN"/>
       </formatter>
   </console-handler>
   <periodic-rotating-file-handler name="FILE" autoflush="true">
       <formatter>
           <named-formatter name="PATTERN"/>
       </formatter>
       <file relative-to="jboss.server.log.dir" path="server.log"/>
       <suffix value=".yyyy-MM-dd"/>
   </periodic-rotating-file-handler>
   <logger category="com.arjuna">
       <level name="WARN"/>
   </logger>
   [...]
   <root-logger>
       <level name="DEBUG"/>
       <handlers>
           <handler name="CONSOLE"/>
           <handler name="FILE"/>
       </handlers>
   </root-logger>
   <formatter name="PATTERN">
       <pattern-formatter pattern="%d{yyyy-MM-dd HH:mm:ss,SSS} %-5p [%c] (%t) %s%e%n"/>
   </formatter>
   <formatter name="COLOR-PATTERN">
       <pattern-formatter pattern="%K{level}%d{HH:mm:ss,SSS} %-5p [%c] (%t) %s%e%n"/>
   </formatter>
</subsystem>
```

Attributes

The root resource contains two notable attributes `add-logging-api-dependencies` and `use-deployment-logging-config`.

logging-api-dependencies

The `add-logging-api-dependencies` controls whether or not the container adds implicit logging API dependencies to your deployments. If set to `true`, the default, all the implicit logging API dependencies are added. If set to `false` the dependencies are not added to your deployments.

deployment-logging-config

The `use-deployment-logging-config` controls whether or not your deployment is scanned for per-deployment logging. If set to `true`, the default, per-deployment logging is enabled. Set to `false` to disable this feature.

deployment Logging

Per-deployment logging allows you to add a logging configuration file to your deployment and have the logging for that deployment configured according to the configuration file. In an EAR the configuration should be in the `META-INF` directory. In a WAR or JAR deployment the configuration file can be in either the `META-INF` or `WEB-INF/classes` directories.

The following configuration files are allowed:

- logging.properties
- jboss-logging.properties
- log4j.properties
- log4j.xml
- jboss-log4j.xml

You can also disable this functionality by changing the `use-deployment-logging-config` attribute to `false`.

Logging Profiles

Logging profiles are like additional logging subsystems. Each logging profile constists of three of the four notable parts listed above: `handler` configurations, `logger` and the `root logger` declarations.

You can assign a logging profile to a deployment via the deployments manifest. Add a `Logging-Profile` entry to the `MANIFEST.MF` file with a value of the logging profile id. For example a logging profile defined on `/subsystem=logging/logging-profile=ejbs` the MANIFEST.MF would look like:

```
Manifest-Version: 1.0
Logging-Profile: ejbs
```

A logging profile can be assigned to any number of deployments. Using a logging profile also allows for runtime changes to the configuration. This is an advantage over the per-deployment logging configuration as the redeploy is not required for logging changes to take affect.

Default Log File Locations

Managed Domain

In a managed domain two types of log files do exist: Controller and server logs. The controller components govern the domain as whole. It's their responsibility to start/stop server instances and execute managed operations throughout the domain. Server logs contain the logging information for a particular server instance. They are co-located with the host the server is running on.

For the sake of simplicity we look at the default setup for managed domain. In this case, both the domain controller components and the servers are located on the same host:

Process	Log File
Host Controller	./domain/log/host-controller.log
Process Controller	./domain/log/process-controller.log
"Server One"	./domain/servers/server-one/log/server.log
"Server Two"	./domain/servers/server-two/log/server.log
"Server Three"	./domain/servers/server-three/log/server.log

Standalone Server

The default log files for a standalone server can be found in the log subdirectory of the distribution:

Process	Log File
Server	./standalone/log/server.log

Filter Expressions

Filter Type	Expression	Description	Parameter(s)	Examples
accept	accept	Accepts all log messages.	None	accept
deny	deny	enies all log messages.	None	deny
not	not(filterExpression)	Accepts a filter as an argument and inverts the returned value.	The expression takes a single filter for it's argument.	not(match("JBAS'
all	all(filterExpressions)	A filter consisting of several filters in a chain. If any filter find the log message to be unloggable, the message will not be logged and subsequent filters will not be checked.	The expression takes a comma delimited list of filters for it's argument.	all(match("JBAS") match("WELD"))
any	any(filterExpressions)	A filter consisting of several filters in a chain. If any filter fins the log message to be loggable, the message will be logged and the subsequent filters will not be checked.	The expression takes a comma delimited list of filters for it's argument.	any(match("JBAS match("WELD"))
levelChange	levelChange(level)	A filter which modifies the log record with a new level.	The expression takes a single string based level for it's argument.	levelChange(WAF

levels	levels(levels)	A filter which includes log messages with a level that is listed in the list of levels.	The expression takes a comma delimited list of string based levels for it's argument.	levels(DEBUG, IN WARN, ERROR)
levelRange	levelRange([minLevel,maxLevel])	A filter which logs records that are within the level range.	The filter expression uses a "[" to indicate a minimum inclusive level and a "]" to indicate a maximum inclusive level. Otherwise use "(" or ")" respectively indicate exclusive. The first argument for the expression is the minimum level allowed, the second argument is the maximum level allowed.	• minimum le be less tha ERROR ar maximum l be greater DEBUG levelRange DEBUG) • minimum le be less tha equal to EF and the ma level must greater tha DEBUG levelRange DEBUG) • minimum le be less tha equal to EF and the ma level must greater or e INFO levelRange INFO]
match	match("pattern")	A regular-expression based filter. The raw unformatted message is used against the pattern.	The expression takes a regular expression for it's argument. match("JBAS\d+")	

substitute	substitute("pattern", "replacement value")	A filter which replaces the first match to the pattern with the replacement value.	The first argument for the expression is the pattern the second argument is the replacement text.	substitute("JBAS"
substituteAll	substituteAll("pattern", "replacement value")	A filter which replaces all matches of the pattern with the replacement value.	The first argument for the expression is the pattern the second argument is the replacement text.	substituteAll("JBA "EAP")

List Log Files and Reading Log Files

Log files can be listed and viewed via management operations. The log files allowed to be viewed are intentionally limited to files that exist in the `jboss.server.log.dir` and are associated with a known file handler. Known file handler types include `file-handler`, `periodic-rotating-file-handler` and `size-rotating-file-handler`. The operations are valid in both standalone and domain modes.

List Log Files

The logging subsystem has a `log-file` resource off the subsystem root resource and off each `logging-profile` resource to list each log file.

CLI command and output

```
[standalone@localhost:9990 /] /subsystem=logging:read-children-names(child-type=log-file)
{
    "outcome" => "success",
    "result" => [
        "server.log",
        "server.log.2014-02-12",
        "server.log.2014-02-13"
    ]
}
```

Read Log File

The `read-log-file` operation is available on each `log-file` resource. This operation has 4 optional parameters.

Name	Description
encoding	the encoding the file should be read in
lines	the number of lines from the file. A value of -1 indicates all lines should be read.
skip	the number of lines to skip before reading.
tail	true to read from the end of the file up or false to read top down.

CLI command and output

```
[standalone@localhost:9990 /] /subsystem=logging/log-file=server.log:read-log-file
{
    "outcome" => "success",
    "result" => [
        "2014-02-14 14:16:48,781 INFO  [org.jboss.as.server.deployment.scanner] (MSC service
thread 1-11) JBAS015012: Started FileSystemDeploymentService for directory
/home/jperkins/servers/wildfly-8.0.0.Final/standalone/deployments",
        "2014-02-14 14:16:48,782 INFO  [org.jboss.as.connector.subsystems.datasources] (MSC
service thread 1-8) JBAS010400: Bound data source [java:jboss/myDs]",
        "2014-02-14 14:16:48,782 INFO  [org.jboss.as.connector.subsystems.datasources] (MSC
service thread 1-15) JBAS010400: Bound data source [java:jboss/datasources/ExampleDS]",
        "2014-02-14 14:16:48,786 INFO  [org.jboss.as.server.deployment] (MSC service thread 1-9)
JBAS015876: Starting deployment of \"simple-servlet.war\" (runtime-name:
\"simple-servlet.war\")",
        "2014-02-14 14:16:48,978 INFO  [org.jboss.ws.common.management] (MSC service thread
1-10) JBWS022052: Starting JBoss Web Services - Stack CXF Server 4.2.3.Final",
        "2014-02-14 14:16:49,160 INFO  [org.wildfly.extension.undertow] (MSC service thread
1-16) JBAS017534: Registered web context: /simple-servlet",
        "2014-02-14 14:16:49,189 INFO  [org.jboss.as.server] (Controller Boot Thread)
JBAS018559: Deployed \"simple-servlet.war\" (runtime-name : \"simple-servlet.war\")",
        "2014-02-14 14:16:49,224 INFO  [org.jboss.as] (Controller Boot Thread) JBAS015961: Http
management interface listening on http://127.0.0.1:9990/management",
        "2014-02-14 14:16:49,224 INFO  [org.jboss.as] (Controller Boot Thread) JBAS015951: Admin
console listening on http://127.0.0.1:9990",
        "2014-02-14 14:16:49,225 INFO  [org.jboss.as] (Controller Boot Thread) JBAS015874:
WildFly 8.0.0.Final \"WildFly\" started in 1906ms - Started 258 of 312 services (90 services are
lazy, passive or on-demand)"
    ]
}
```

FAQ

Why is there a logging.properties file?

You may have noticed that there is a `logging.properties` file in the configuration directory. This is logging configuration is used when the server boots up until the logging subsystem kicks in. If the logging subsystem is not included in your configuration, then this would act as the logging configuration for the entire server.

The `logging.properties` file is overwritten at boot and with each change to the logging subsystem. Any changes made to the file are not persisted. Any changes made to the XML configuration or via management operations will be persisted to the `logging.properties` file and used on the next boot.

5.7.5 Web (Undertow)

Web subsystem was replaced in WildFly 8 with Undertow.

There are two main parts to the undertow subsystem, which are server and Servlet container configuration, as well as some ancillary items. Advanced topics like load balancing and failover are covered on the "High Availability Guide". The default configuration does is suitable for most use cases and provides reasonable performance settings.

Required extension:

```
<extension module="org.wildfly.extension.undertow" />
```

Basic subsystem configuration example:

```
<subsystem xmlns="urn:jboss:domain:undertow:1.0">
        <buffer-caches>
            <buffer-cache name="default" buffer-size="1024" buffers-per-region="1024"
max-regions="10"/>
        </buffer-caches>
        <server name="default-server">
            <http-listener name="default" socket-binding="http" />
            <host name="default-host" alias="localhost">
                <location name="/" handler="welcome-content" />
            </host>
        </server>
        <servlet-container name="default" default-buffer-cache="default"
stack-trace-on-error="local-only" >
            <jsp-config/>
            <persistent-sessions/>
        </servlet-container>
        <handlers>
            <file name="welcome-content" path="${jboss.home.dir}/welcome-content"
directory-listing="true"/>
        </handlers>
    </subsystem>
```

Dependencies on other subsystems:

IO Subsystem

Buffer cache configuration

The buffer cache is used for caching content, such as static files. Multiple buffer caches can be configured, which allows for separate servers to use different sized caches.

Buffers are allocated in regions, and are of a fixed size. If you are caching many small files then using a smaller buffer size will be better.

The total amount of space used can be calculated by multiplying the buffer size by the number of buffers per region by the maximum number of regions.

```
<buffer-caches>
    <buffer-cache name="default" buffer-size="1024" buffers-per-region="1024" max-regions="10"/>
  </buffer-caches>
```

Attribute	Description
buffer-size	The size of the buffers. Smaller buffers allow space to be utilised more effectively
buffers-per-region	The numbers of buffers per region
max-regions	The maximum number of regions. This controls the maximum amount of memory that can be used for caching

Server configuration

A server represents an instance of Undertow. Basically this consists of a set of connectors and some configured handlers.

```
<server name="default-server" default-host="default-host" servlet-container="default" >
```

Attribute	Description
default-host	the virtual host that will be used if an incoming request as no Host: header
servlet-container	the servlet container that will be used by this server, unless is is explicitly overriden by the deployment

Connector configuration

Undertow provides HTTP, HTTPS and AJP connectors, which are configured per server.

Common settings

The following settings are common to all connectors:

Attribute	Description
socket-binding	The socket binding to use. This determines the address and port the listener listens on.
worker	A reference to an XNIO worker, as defined in the IO subsystem. The worker that is in use controls the IO and blocking thread pool.
buffer-pool	A reference to a buffer pool as defined in the IO subsystem. These buffers are used internally to read and write requests. In general these should be at least 8k, unless you are in a memory constrained environment.
enabled	If the connector is enabled.
max-post-size	The maximum size of incoming post requests that is allowed.
buffer-pipelined-data	If responses to HTTP pipelined requests should be buffered, and send out in a single write. This can improve performance if HTTP pipe lining is in use and responses are small.
max-header-size	The maximum size of a HTTP header block that is allowed. Responses with to much data in their header block will have the request terminated and a bad request response send.
max-parameters	The maximum number of query or path parameters that are allowed. This limit exists to prevent hash collision based DOS attacks.
max-headers	The maximum number of headers that are allowed. This limit exists to prevent hash collision based DOS attacks.
max-cookies	The maximum number of cookies that are allowed. This limit exists to prevent hash collision based DOS attacks.
allow-encoded-slash	Set this to true if you want the server to decode percent encoded slash characters. This is probably a bad idea, as it can have security implications, due to different servers interpreting the slash differently. Only enable this if you have a legacy application that requires it.
decode-url	If the URL should be decoded. If this is not set to true then percent encoded characters in the URL will be left as is.
url-charset	The charset to decode the URL to.
always-set-keep-alive	If the 'Connection: keep-alive' header should be added to all responses, even if not required by spec.
disallowed-methods	A comma separated list of HTTP methods that are not allowed. HTTP TRACE is disabled by default.

HTTP Connector

```
<http-listener name="default" socket-binding="http"  />
```

Attribute	Description
certificate-forwarding	If this is set to true then the HTTP listener will read a client certificate from the SSL_CLIENT_CERT header. This allows client cert authentication to be used, even if the server does not have a direct SSL connection to the end user. This should only be enabled for servers behind a proxy that has been configured to always set these headers.
redirect-socket	The socket binding to redirect requests that require security too.
proxy-address-forwarding	If this is enabled then the X-Forwarded-For and X-Forwarded-Proto headers will be used to determine the peer address. This allows applications that are behind a proxy to see the real address of the client, rather than the address of the proxy.

HTTPS listener

Https listener provides secure access to the server. The most important configuration option is security realm which defines SSL secure context.

```
<https-listener name="default" socket-binding="https" security-realm="ssl-realm" />
```

Attribute	Description
security-realm	The security realm to use for the SSL configuration. See Security realm examples for how to configure it: Examples
verify-client	One of either NOT_REQUESTED, REQUESTED or REQUIRED. If client cert auth is in use this should be either REQUESTED or REQUIRED.
enabled-cipher-suites	A list of cypher suit names that are allowed.

AJP listener

```
<ajp-listener name="default" socket-binding="ajp" />
```

Host configuration

The host element corresponds to a virtual host.

Attribute	Description
name	The virtual host name
alias	A whitespace separated list of additional host names that should be matched
default-web-module	The name of a deployment that should be used to serve up requests that do not match anything.

Servlet container configuration

The servlet-container element corresponds to an instance of an Undertow Servlet container. Most servers will only need a single servlet container, however there may be cases where it makes sense to define multiple containers (in particular if you want applications to be isolated, so they cannot dispatch to each other using the RequestDispatcher. You can also use multiple Servlet containers to serve different applications from the same context path on different virtual hosts).

Attribute	Description
allow-non-standard-wrappers	The Servlet specification requires applications to only wrap the request/response using wrapper classes that extend from the ServletRequestWrapper and ServletResponseWrapper classes. If this is set to true then this restriction is relaxed.
default-buffer-cache	The buffer cache that is used to cache static resources in the default Servlet.
stack-trace-on-error	Can be either all, none, or local-only. When set to none Undertow will never display stack traces. When set to All Undertow will always display them (not recommended for production use). When set to local-only Undertow will only display them for requests from local addresses, where there are no headers to indicate that the request has been proxied. Note that this feature means that the Undertow error page will be displayed instead of the default error page specified in web.xml.
default-encoding	The default encoding to use for requests and responses.
use-listener-encoding	If this is true then the default encoding will be the same as that used by the listener that received the request.

JSP configuration

Session Cookie Configuration

This allows you to change the attributes of the session cookie.

Attribute	Description
name	The cookie name
domain	The cookie domain
comment	The cookie comment
http-only	If the cookie is HTTP only
secure	If the cookie is marked secure
max-age	The max age of the cookie

Persistent Session Configuration

Persistent sessions allow session data to be saved across redeploys and restarts. This feature is enabled by adding the persistent-sessions element to the server config. This is mostly intended to be a development time feature.

If the path is not specified then session data is stored in memory, and will only be persistent across redeploys, rather than restarts.

Attribute	Description
path	The path to the persistent sessions data
relative-to	The location that the path is relevant to

5.7.6 Messaging

The JMS server configuration is done through the *messaging-activemq* subsystem. In this chapter we are going outline the frequently used configuration options. For a more detailed explanation please consult the Artemis user guide (See "Component Reference").

Required Extension

The configuration options discussed in this section assume that the the `org.wildfly.extension.messaging-activemq` extension is present in your configuration. This extension is not included in the standard `standalone.xml` and `standalone-ha.xml` configurations included in the WildFly distribution. It is, however, included with the `standalone-full.xml` and `standalone-full-ha.xml` configurations.

You can add the extension to a configuration without it either by adding an `<extension module="org.wildfly.extension.messaging-activemq"/>` element to the xml or by using the following CLI operation:

```
[standalone@localhost:9990 /]/extension=org.wildfly.extension.messaging-activemq:add
```

Connectors

There are three kind of connectors that can be used to connect to WildFly JMS Server

- `in-vm-connector` can be used by a local client (i.e. one running in the same JVM as the server)
- `remote-connector` can be used by a remote client (and uses Netty over TCP for the communication)
- `http-connector` can be used by a remote client (and uses Undertow Web Server to upgrade from a HTTP connection)

JMS Connection Factories

There are three kinds of *basic* JMS `connection-factory` that depends on the type of connectors that is used.

There is also a `pooled-connection-factory` which is special in that it is essentially a configuration facade for *both* the inbound and outbound connectors of the the Artemis JCA Resource Adapter. An MDB can be configured to use a `pooled-connection-factory` (e.g. using `@ResourceAdapter`). In this context, the MDB leverages the *inbound connector* of the Artemis JCA RA. Other kinds of clients can look up the pooled-connection-factory in JNDI (or inject it) and use it to send messages. In this context, such a client would leverage the *outbound connector* of the Artemis JCA RA. A `pooled-connection-factory` is also special because:

- It is only available to local clients, although it can be configured to point to a remote server.
- As the name suggests, it is pooled and therefore provides superior performance to the clients which are able to use it. The pool size can be configured via the `max-pool-size` and `min-pool-size` attributes.
- It should only be used to *send* (i.e. produce) messages when looked up in JNDI or injected.
- It can be configured to use specific security credentials via the `user` and `password` attributes. This is useful if the remote server to which it is pointing is secured.
- Resources acquired from it will be automatically enlisted any on-going JTA transaction. If you want to send a message from an EJB using CMT then this is likely the connection factory you want to use so the send operation will be atomically committed along with the rest of the EJB's transaction operations.

To be clear, the *inbound connector* of the Artemis JCA RA (which is for consuming messages) is only used by MDBs and other JCA-based components. It is not available to traditional clients.

Both a `connection-factory` and a `pooled-connection-factory` reference a `connector` declaration.

A `remote-connector` is associated with a `socket-binding` which tells the client using the `connection-factory` where to connect.

- A `connection-factory` referencing a `remote-connector` is suitable to be used by a *remote* client to send messages to or receive messages from the server (assuming the connection-factory has an appropriately exported `entry`).
- A `pooled-connection-factory` looked up in JNDI or injected which is referencing a `remote-connector` is suitable to be used by a *local* client to send messages to a remote server granted the `socket-binding` references an `outbound-socket-binding` pointing to the remote server in question.
- A `pooled-connection-factory` used by an MDB which is referencing a `remote-connector` is suitable to consume messages from a remote server granted the `socket-binding` references an `outbound-socket-binding` pointing to the remote server in question.

An `in-vm-connector` is associated with a `server-id` which tells the client using the `connection-factory` where to connect (since multiple Artemis servers can run in a single JVM).

- A `connection-factory` referencing an `in-vm-connector` is suitable to be used by a *local* client to either send messages to or receive messages from a local server.
- A `pooled-connection-factory` looked up in JNDI or injected which is referencing an `in-vm-connector` is suitable to be used by a *local* client only to send messages to a local server.
- A `pooled-connection-factory` used by an MDB which is referencing an `in-vm-connector` is suitable only to consume messages from a local server.

A `http-connector` is associated with the `socket-binding` that represents the HTTP socket (by default, named `http`).

- A `connection-factory` referencing a `http-connector` is suitable to be used by a remote client to send messages to or receive messages from the server by connecting to its HTTP port before upgrading to the messaging protocol.
- A `pooled-connection-factory` referencing a `http-connector` is suitable to be used by a local client to send messages to a remote server granted the `socket-binding` references an `outbound-socket-binding` pointing to the remote server in question.
- A `pooled-connection-factory` used by an MDB which is referencing a `http-connector` is suitable only to consume messages from a remote server granted the `socket-binding` references an `outbound-socket-binding` pointing to the remote server in question.

The `entry` declaration of a `connection-factory` or a `pooled-connection-factory` specifies the JNDI name under which the factory will be exposed. Only JNDI names bound in the `"java:jboss/exported"` namespace are available to remote clients. If a `connection-factory` has an entry bound in the `"java:jboss/exported"` namespace a remote client would look-up the `connection-factory` using the text *after* `"java:jboss/exported"`. For example, the "`RemoteConnectionFactory`" is bound by default to `"java:jboss/exported/jms/RemoteConnectionFactory"` which means a remote client would look-up this `connection-factory` using "`jms/RemoteConnectionFactory`". A `pooled-connection-factory` should *not* have any `entry` bound in the "`java:jboss/exported`" namespace because a `pooled-connection-factory` is not suitable for remote clients.

Since JMS 2.0, a default JMS connection factory is accessible to EE application under the JNDI name `java:comp/DefaultJMSConnectionFactory.` WildFly messaging subsystem defines a `pooled-connection-factory` that is used to provide this default connection factory. Any parameter change on this `pooled-connection-factory` will be take into account by any EE application looking the default JMS provider under the JNDI name `java:comp/DefaultJMSConnectionFactory.`

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:1.0">
    <server name="default">
        [...]
        <http-connector name="http-connector"
                        socket-binding="http"
                        endpoint="http-acceptor" />
        <http-connector name="http-connector-throughput"
                        socket-binding="http"
                        endpoint="http-acceptor-throughput">
            <param name="batch-delay"
                   value="50"/>
        </http-connector>
        <in-vm-connector name="in-vm"
                         server-id="0"/>
      [...]
      <connection-factory name="InVmConnectionFactory"
                          connectors="in-vm"
                          entries="java:/ConnectionFactory" />
      <pooled-connection-factory name="activemq-ra"
                          transaction="xa"
                          connectors="in-vm"
                          entries="java:/JmsXA java:jboss/DefaultJMSConnectionFactory"/>
      [...]
   </server>
</subsystem>
```

(See standalone/configuration/standalone-full.xml)

JMS Queues and Topics

JMS queues and topics are sub resources of the messaging-actively subsystem. One can define either a `jms-queue` or `jms-topic`. Each destination *must* be given a `name` and contain at least one entry in its `entries` element (separated by whitespace).

Each entry refers to a JNDI name of the queue or topic. Keep in mind that any `jms-queue` or `jms-topic` which needs to be accessed by a remote client needs to have an entry in the "java:jboss/exported" namespace. As with connection factories, if a `jms-queue` or or `jms-topic` has an entry bound in the "java:jboss/exported" namespace a remote client would look it up using the text *after* `"java:jboss/exported`". For example, the following `jms-queue` "testQueue" is bound to "java:jboss/exported/jms/queue/test" which means a remote client would look-up this {{kms-queue} using "jms/queue/test". A local client could look it up using "java:jboss/exported/jms/queue/test", "java:jms/queue/test", or more simply "jms/queue/test":

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:1.0">
    <server name="default">
    [...]
    <jms-queue name="testQueue"
               entries="jms/queue/test java:jboss/exported/jms/queue/test" />
    <jms-topic name="testTopic"
               entries="jms/topic/test java:jboss/exported/jms/topic/test" />
</subsystem>
```

(See standalone/configuration/standalone-full.xml)

JMS endpoints can easily be created through the CLI:

```
[standalone@localhost:9990 /] jms-queue add --queue-address=myQueue --entries=queues/myQueue
```

```
[standalone@localhost:9990 /]
/subsystem=messaging-activemq/server=default/jms-queue=myQueue:read-resource
{
    "outcome" => "success",
    "result" => {
        "durable" => true,
        "entries" => ["queues/myQueue"],
        "selector" => undefined
    }
}
```

A number of additional commands to maintain the JMS subsystem are available as well:

```
[standalone@localhost:9990 /] jms-queue --help --commands
add
...
remove
To read the description of a specific command execute 'jms-queue command_name --help'.
```

Dead Letter & Redelivery

Some of the settings are applied against an address wild card instead of a specific messaging destination. The dead letter queue and redelivery settings belong into this group:

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:1.0">
   <server name="default">
      [...]
      <address-setting name="#"
                       dead-letter-address="jms.queue.DLQ"
                       expiry-address="jms.queue.ExpiryQueue"
                       [...] />
```

(See standalone/configuration/standalone-full.xml)

Security Settings for Artemis addresses and JMS destinations

Security constraints are matched against an address wildcard, similar to the DLQ and redelivery settings.

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:1.0">
   <server name="default">
      [...]
      <security-setting name="#">
          <role name="guest"
                send="true"
                consume="true"
                create-non-durable-queue="true"
                delete-non-durable-queue="true"/>
```

(See standalone/configuration/standalone-full.xml)

Security Domain for Users

By default, Artemis will use the "`other`" JAAS security domain. This domain is used to authenticate users making connections to Artemis and then they are authorized to perform specific functions based on their role(s) and the `security-settings` described above. This domain can be changed by using the `security-domain`, e.g.:

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:1.0">
   <server name="default">
       <security domain="mySecurityDomain" />
      [...]
```

Using the Elytron Subsystem

You can also use the elytron subsystem to secure the messaging-activemq subsystem.

To use an Elytron security domain:

1. Undefine the legacy security domain.

```
/subsystem=messaging-activemq/server=default:undefine-attribute(name=security-domain)
```

2. Set an Elytron security domain.

```
/subsystem=messaging-activemq/server=default:write-attribute(name=elytron-domain,
value=myElytronSecurityDomain)
```

You can only define either `security-domain` or `elytron-domain`, but you cannot have both defined at the same time. If neither is defined, WildFly will use the `security-domain` default value of `other`, which maps to the `other` legacy security domain.

Cluster Authentication

If the Artemis server is configured to be clustered, it will use the `cluster` 's `user` and `password` attributes to connect to other Artemis nodes in the cluster.

If you do not change the default value of <cluster-password>, Artemis will fail to authenticate with the error:

```
HQ224018: Failed to create session: HornetQExceptionerrorType=CLUSTER_SECURITY_EXCEPTION
message=HQ119099: Unable to authenticate cluster user: HORNETQ.CLUSTER.ADMIN.USER
```

To prevent this error, you must specify a value for `<cluster-password>`. It is possible to encrypt this value by following this guide.

Alternatively, you can use the system property jboss.messaging.cluster.password to specify the cluster password from the command line.

Deployment of -jms.xml files

Starting with WildFly 8, you have the ability to deploy a -jms.xml file defining JMS destinations, e.g.:

```
<?xml version="1.0" encoding="UTF-8"?>
<messaging-deployment xmlns="urn:jboss:messaging-activemq-deployment:1.0">
   <server name="default">
      <jms-destinations>
         <jms-queue name="sample">
            <entry name="jms/queue/sample"/>
            <entry name="java:jboss/exported/jms/queue/sample"/>
         </jms-queue>
      </jms-destinations>
   </server>
</messaging-deployment>
```

This feature **is primarily intended for development** as destinations deployed this way can not be managed with any of the provided management tools (e.g. console, CLI, etc).

JMS Bridge

The function of a JMS bridge is to consume messages from a source JMS destination, and send them to a target JMS destination. Typically either the source or the target destinations are on different servers.
The bridge can also be used to bridge messages from other non Artemis JMS servers, as long as they are JMS 1.1 compliant.

The JMS Bridge is provided by the Artemis project. For a detailed description of the available configuration properties, please consult the project documentation.

Modules for other messaging brokers

Source and target JMS resources (destination and connection factories) are looked up using JNDI.
If either the source or the target resources are managed by another messaging server than WildFly, the required client classes must be bundled in a module. The name of the module must then be declared when the JMS Bridge is configured.

The use of a JMS bridges with any messaging provider will require to create a module containing the jar of this provider.

Let's suppose we want to use an hypothetical messaging provider named AcmeMQ. We want to bridge messages coming from a source AcmeMQ destination to a target destination on the local WildFly messaging server. To lookup AcmeMQ resources from JNDI, 2 jars are required, acmemq-1.2.3.jar, mylogapi-0.0.1.jar (please note these jars do not exist, this is just for the example purpose). We must *not* include a JMS jar since it will be provided by a WildFly module directly.

To use these resources in a JMS bridge, we must bundle them in a WildFly module:

in JBOSS_HOME/modules, we create the layout:

```
modules/
`-- org
    `-- acmemq
        `-- main
            |-- acmemq-1.2.3.jar
            |-- mylogapi-0.0.1.jar
            `-- module.xml
```

We define the module in `module.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<module xmlns="urn:jboss:module:1.1" name="org.acmemq">
    <properties>
        <property name="jboss.api" value="private"/>
    </properties>


    <resources>
        <!-- insert resources required to connect to the source or target   -->
        <!-- messaging brokers if it not another WildFly instance           -->
        <resource-root path="acmemq-1.2.3.jar" />
        <resource-root path="mylogapi-0.0.1.jar" />
    </resources>


    <dependencies>
       <!-- add the dependencies required by JMS Bridge code               -->
       <module name="javax.api" />
       <module name="javax.jms.api" />
       <module name="javax.transaction.api"/>
       <module name="org.jboss.remote-naming"/>
       <!-- we depend on org.apache.activemq.artemis module since we will send messages to  -->
       <!-- the Artemis server embedded in the local WildFly instance       -->
       <module name="org.apache.activemq.artemis" />
    </dependencies>
</module>
```

Configuration

A JMS bridge is defined inside a `jms-bridge` section of the `messaging-activemq` subsystem in the XML configuration files.

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:1.0">
   <jms-bridge name="myBridge" module="org.acmemq">
      <source connection-factory="ConnectionFactory"
              destination="sourceQ"
              user="user1"
              password="pwd1"
              quality-of-service="AT_MOST_ONCE"
              failure-retry-interval="500"
              max-retries="1"
              max-batch-size="500"
              max-batch-time="500"
              add-messageID-in-header="true">
         <source-context>
            <property name="java.naming.factory.initial"
                      value="org.acmemq.jndi.AcmeMQInitialContextFactory"/>
            <property name="java.naming.provider.url"
                      value="tcp://127.0.0.1:9292"/>
         </source-context>
      </source>
      <target connection-factory"/jms/invmTargetCF"
              destination="/jms/targetQ" />
      </target>
   </jms-bridge>
</subsystem>
```

The `source` and `target` sections contain the name of the JMS resource (`connection-factory` and `destination`) that will be looked up in JNDI.
It optionally defines the `user` and `password` credentials. If they are set, they will be passed as arguments when creating the JMS connection from the looked up ConnectionFactory.
It is also possible to define JNDI context properties in the `source-context` and `target-context` sections. If these sections are absent, the JMS resources will be looked up in the local WildFly instance (as it is the case in the `target` section in the example above).

Management commands

A JMS Bridge can also be managed using the WildFly command line interface:

```
[standalone@localhost:9990 /] /subsystem=messaging/jms-bridge=myBridge/:add(module="org.acmemq",
\
      source-destination="sourceQ",
\
      source-connection-factory="ConnectionFactory",
\
      source-user="user1",
\
      source-password="pwd1",
\
      source-context={"java.naming.factory.initial" =>
"org.acmemq.jndi.AcmeMQInitialContextFactory", \
                     "java.naming.provider.url" => "tcp://127.0.0.1:9292"},
\
      target-destination="/jms/targetQ",
\
      target-connection-factory="/jms/invmTargetCF",
\
      quality-of-service=AT_MOST_ONCE,
\
      failure-retry-interval=500,
\
      max-retries=1,
\
      max-batch-size=500,
\
      max-batch-time=500,
\
      add-messageID-in-header=true)
{"outcome" => "success"}
```

You can also see the complete JMS Bridge resource description from the CLI:

```
[standalone@localhost:9990 /] /subsystem=messaging/jms-bridge=*/:read-resource-description
{
    "outcome" => "success",
    "result" => [{
        "address" => [
            ("subsystem" => "messaging"),
            ("jms-bridge" => "*")
        ],
        "outcome" => "success",
        "result" => {
            "description" => "A JMS bridge instance.",
            "attributes" => {
                ...
        }
    }]
}
```

Component Reference

The messaging-activemq subsystem is provided by the Artemis project. For a detailed description of the available configuration properties, please consult the project documentation.

- Artemis Homepage: http://activemq.apache.org/artemis/
- Artemis User Documentation: http://activemq.apache.org/artemis/docs.html

5.7.7 Security

The security subsystem is the subsystem that brings the security services provided by PicketBox to the WildFly 8 server instances.

If you are looking to secure the management interfaces for the management of the domain then you should read the Securing the Management Interfaces chapter as the management interfaces themselves are not run within a WildFly process so use a custom configuration.

Structure of the Security Subsystem

When deploying applications to WildFly most of the time it is likely that you would be deploying a web application or EJBs and just require a security domain to be defined with login modules to verify the users identity, this chapter aims to provide additional detail regarding the architecture and capability of the security subsystem however if you are just looking to define a security domain and leave the rest to the container please jump to the security-domains section.

The security subsystem operates by using a security context associated with the current request, this security context then makes available to the relevant container a number of capabilities from the configured security domain, the capabilities exposed are an authentication manager, an authorization manager, an audit manager and a mapping manager.

Authentication Manager

The authentication manager is the component that performs the actual authentication taking the declared users identity and their credential so that the login context for the security domain can be used to 'login' the user using the configured login module or modules.

Authorization Manager

The authorization manager is a component which can be obtained by the container from the current security context to either obtain information about a users roles or to perform an authorization check against a resource for the currently authenticated user.

Audit Manager

The audit manager from the security context is the component that can be used to log audit events in relation to the security domain.

Mapping Manager

The mapping manager can be used to assign additional principals, credentials, roles or attributes to the authenticated subject.

Security Subsystem Configuration

By default a lot of defaults have already been selected for the security subsystem and unless there is a specific implementation detail you need to change, these defaults should not require modification. This chapter describes all of the possible configuration attributes for completeness but do keep in mind that not all will need to be changed.

The security subsystem is enabled by default by the addition of the following extension: -

<extension module="org.jboss.as.security"/>

The namespace used for the configuration of the security subsystem is urn:jboss:domain:security:1.0, the configuration is defined within the <subsystem> element from this namespace.

The <subsystem> element can optionally contain the following child elements.

- security-management
- subject-factory
- security-domains
- security-properties

security-management

This element is used to override some of the high level implementation details of the PicketBox implementation if you have a need to change some of this behaviour.

The element can have any or the following attributes set, all of which are optional.

authentication-manager-class-name	Specifies the AuthenticationManager implementation class name to use.
deep-copy-subject-mode	Sets the copy mode of subjects done by the security managers to be deep copies that makes copies of the subject principals and credentials if they are cloneable. It should be set to true if subject include mutable content that can be corrupted when multiple threads have the same identity and cache flushes/logout clearing the subject in one thread results in subject references affecting other threads. Default value is "false".
default-callback-handler-class-name	Specifies a global class name for the CallbackHandler implementation to be used with login modules.
authorization-manager-class-name	Attribute specifies the AuthorizationManager implementation class name to use.
audit-manager-class-name	Specifies the AuditManager implementation class name to use.
identity-trust-manager-class-name	Specifies the IdentityTrustManager implementation class name to use.
mapping-manager-class-name	Specifies the MappingManager implementation class name to use.

subject-factory

The subject factory is responsible for creating subject instances, this also makes use of the authentication manager to actually verify the caller. It is used mainly by JCA components to establish a subject. It is not likely this would need to be overridden but if it is required the "subject-factory-class-name" attribute can be specified on the subject-factory element.

security-domains

This portion of the configuration is where the bulk of the security subsystem configuration will actually take place for most administrators, the security domains contain the configuration which is specific to a deployment.

The security-domains element can contain numerous <security-domain> definitions, a security-domain can have the following attributes set:

name	The unique name of this security domain.
~~extends~~	Although version 1.0 of the security subsystem schema contained an 'extends' attribute, security domain inheritance is not supported and this attribute should not be used.
cache-type	The type of authentication cache to use with this domain. If this attribute is removed no cache will be used. Allowed values are "default" or "infinispan"

The following elements can then be set within the security-domain to configure the domain behaviour.

authentication

The authentication element is used to hold the list of login modules that will be used for authentication when this domain is used, the structure of the login-module element is:

```
<login-module code="..." flag="..." module="...">
  <module-option name="..." value="..."/>
</login-module>
```

The code attribute is used to specify the implementing class of the login module which can either be the full class name or one of the abbreviated names from the following list:

Code	Classname
Client	`org.jboss.security.ClientLoginModule`
Certificate	`org.jboss.security.auth.spi.BaseCertLoginModule`
CertificateUsers	`org.jboss.security.auth.spi.BaseCertLoginModule`
CertificateRoles	`org.jboss.security.auth.spi.CertRolesLoginModule`
Database	`org.jboss.security.auth.spi.DatabaseServerLoginModule`
DatabaseCertificate	`org.jboss.security.auth.spi.DatabaseCertLoginModule`
DatabaseUsers	`org.jboss.security.auth.spi.DatabaseServerLoginModule`
Identity	`org.jboss.security.auth.spi.IdentityLoginModule`
Ldap	`org.jboss.security.auth.spi.LdapLoginModule`
LdapExtended	`org.jboss.security.auth.spi.LdapExtLoginModule`
RoleMapping	`org.jboss.security.auth.spi.RoleMappingLoginModule`
RunAs	`org.jboss.security.auth.spi.RunAsLoginModule`
Simple	`org.jboss.security.auth.spi.SimpleServerLoginModule`
ConfiguredIdentity	`org.picketbox.datasource.security.ConfiguredIdentityLoginModule`
SecureIdentity	`org.picketbox.datasource.security.SecureIdentityLoginModule`
PropertiesUsers	`org.jboss.security.auth.spi.PropertiesUsersLoginModule`
SimpleUsers	`org.jboss.security.auth.spi.SimpleUsersLoginModule`
LdapUsers	`org.jboss.security.auth.spi.LdapUsersLoginModule`
Kerberos	`com.sun.security.auth.module.Krb5LoginModule`
SPNEGOUsers	`org.jboss.security.negotiation.spnego.SPNEGOLoginModule`
AdvancedLdap	`org.jboss.security.negotiation.AdvancedLdapLoginModule`
AdvancedADLdap	`org.jboss.security.negotiation.AdvancedADLoginModule`
UsersRoles	`org.jboss.security.auth.spi.UsersRolesLoginModule`

The module attribute specifies the name of the JBoss Modules module from which the class specified by the code attribute should be loaded. Specifying it is not necessary if one of the abbreviated names in the above list is used.

The flag attribute is used to specify the JAAS flag for this module and should be one of required, requisite, sufficient, or optional.

The module-option element can be repeated zero or more times to specify the module options as required for the login module being configured. It requires the name and value attributes.

See Authentication Modules for further details on the various modules listed above.

authentication-jaspi

The authentication-jaspi is used to configure a Java Authentication SPI (JASPI) provider as the authentication mechanism. A security domain can have either a <authentication> or a <authentication-jaspi> element, but not both. We set up JASPI by configuring one or more login modules inside the login-module-stack element and setting up an authentication module. Here is the structure of the authentication-jaspi element:

```
<login-module-stack name="...">
  <login-module code="..." flag="..." module="...">
    <module-option name="..." value="..."/>
  </login-module>
</login-module-stack>
<auth-module code="..." login-module-stack-ref="...">
  <module-option name="..." value="..."/>
</auth-module>
```

The login-module-stack-ref attribute value must be the name of the login-module-stack element to be used. The sub-element login-module is configured just like in the authentication part

authorization

Authorization in the AS container is normally done with RBAC (role based access control) but there are situations where a more fine grained authorization policy is required. The authorization element allows definition of different authorization modules to used, such that authorization can be checked with JACC (Java Authorization Contract for Containers) or XACML (eXtensible Access Control Markup Language). The structure of the authorization element is:

```
<policy-module code="..." flag="..." module="...">
  <module-option name="..." value="..."/>
</policy-module>
```

The code attribute is used to specify the implementing class of the policy module which can either be the full class name or one of the abbreviated names from the following list:

Code	Classname
DenyAll	`org.jboss.security.authorization.modules.AllDenyAuthorizationModule`
PermitAll	`org.jboss.security.authorization.modules.AllPermitAuthorizationModule`
Delegating	`org.jboss.security.authorization.modules.DelegatingAuthorizationModule`
Web	`org.jboss.security.authorization.modules.WebAuthorizationModule`
JACC	`org.jboss.security.authorization.modules.JACCAuthorizationModule`
XACML	`org.jboss.security.authorization.modules.XACMLAuthorizationModule`

The module attribute specifies the name of the JBoss Modules module from which the class specified by the code attribute should be loaded. Specifying it is not necessary if one of the abbreviated names in the above list is used.

The flag attribute is used to specify the JAAS flag for this module and should be one of required, requisite, sufficient, or optional.

The module-option element can be repeated zero or more times to specify the module options as required for the login module being configured. It requires the name and value attributes.

mapping

The mapping element defines additional mapping of principals, credentials, roles and attributes for the subject. The structure of the mapping element is:

```
<mapping-module type="..."code="..." module="...">
  <module-option name="..." value="..."/>
</mapping-module>
```

The type attribute reflects the type of mapping of the provider and should be one of principal, credential, role or attribute. By default "role" is the type used if the attribute is not set.

The code attribute is used to specify the implementing class of the login module which can either be the full class name or one of the abbreviated names from the following list:

Code	Classname
PropertiesRoles	`org.jboss.security.mapping.providers.role.PropertiesRolesMappingP`
SimpleRoles	`org.jboss.security.mapping.providers.role.SimpleRolesMappingProvi`
DeploymentRoles	`org.jboss.security.mapping.providers.DeploymentRolesMappingProvid`
DatabaseRoles	`org.jboss.security.mapping.providers.role.DatabaseRolesMappingPro`
LdapRoles	`org.jboss.security.mapping.providers.role.LdapRolesMappingProvide`

The module attribute specifies the name of the JBoss Modules module from which the class specified by the code attribute should be loaded. Specifying it is not necessary if one of the abbreviated names in the above list is used.

The module-option element can be repeated zero or more times to specify the module options as required for the login module being configured. It requires the name and value attributes.

audit

The audit element can be used to define a custom audit provider. The default implementation used is `org.jboss.security.audit.providers.LogAuditProvider`. The structure of the audit element is:

```
<provider-module code="..." module="...">
  <module-option name="..." value="..."/>
</provider-module>
```

The code attribute is used to specify the implementing class of the provider module.

The module attribute specifies the name of the JBoss Modules module from which the class specified by the code attribute should be loaded.

The module-option element can be repeated zero or more times to specify the module options as required for the login module being configured. It requires the name and value attributes.

jsse

The jsse element defines configuration for keystores and trustores that can be used for SSL context configuration or for certificate storing/retrieving.

The set of attributes (all of them optional) of this element are:

keystore-password	Password of the keystore
keystore-type	Type of the keystore. By default it's "JKS"
keystore-url	`URL` where the keystore file can be found
keystore-provider	`Provider` of the keystore. The default JDK provider for the keystore type is used if this attribute is null
keystore-provider-argument	String that can be passed as the argument of the keystore `Provider` constructor
key-manager-factory-algorithm	Algorithm of the `KeyManagerFactory`. The default JDK algorithm of the key manager factory is used if this attribute is null
key-manager-factory-provider	`Provider` of the `KeyManagerFactory`. The default JDK provider for the key manager factory algorithm is used if this attribute is null
truststore-password	Password of the truststore
truststore-type	Type of the truststore. By deafult it's "JKS"
truststore-url	`URL` where the truststore file can be found
truststore-provider	`Provider` of the truststore. The default JDK provider for the truststore type is used if this attribute is null
truststore-provider-argument	String that can be passed as the argument of the truststore `Provider` constructor
trust-manager-factory-algorithm	Algorithm of the `TrustManagerFactory`. The default JDK algorithm of the trust manager factory is used if this attribute is null
trust-manager-factory-provider	`Provider` of the `TrustManagerFactory`. The default JDK provider for the trust manager factory algorithm is used if this attribute is null
client-alias	Alias of the keystore to be used when creating client side SSL sockets
server-alias	Alias of the keystore to be used when creating server side SSL sockets
service-auth-token	Validation token to enable third party services to retrieve a keystore `Key`. This is typically used to retrieve a private key for signing purposes
client-auth	Flag to indicate if the server side SSL socket should require client authentication. Default is "false"
cipher-suites	Comma separated list of cipher suites to be used by a `SSLContext`
protocols	Comma separated list of SSL protocols to be used by a `SSLContext`

The optional additional-properties element can be used to include other options. The structure of the jsse element is:

```
<jsse keystore-url="..." keystore-password="..." keystore-type="..." keystore-provider="..."
keystore-provider-argument="..." key-manager-factory-algorithm="..."
key-manager-factory-provider="..." truststore-url="..." truststore-password="..."
truststore-type="..." truststore-provider="..." truststore-provider-argument="..."
trust-manager-factory-algorithm="..." trust-manager-factory-provider="..." client-alias="..."
server-alias="..." service-auth-token="..." client-auth="..." cipher-suites="..."
protocols="...">
  <additional-properties>x=y
    a=b
  </additional-properties>
</jsse>
```

security-properties

This element is used to specify additional properties as required by the security subsystem, properties are specified in the following format:

```
<security-properties>
  <property name="..." value="..."/>
</security-properties>
```

The property element can be repeated as required for as many properties need to be defined.

Each property specified is set on the `java.security.Security` class.

5.7.8 Web services

JBossWS components are provided to the application server through the webservices subsystem. JBossWS components handle the processing of WS endpoints. The subsystem supports the configuration of published endpoint addresses, and endpoint handler chains. A default webservice subsystem is provided in the server's domain and standalone configuration files.

Structure of the webservices subsystem

Published endpoint address

JBossWS supports the rewriting of the `<soap:address>` element of endpoints published in WSDL contracts. This feature is useful for controlling the server address that is advertised to clients for each endpoint.

The following elements are available and can be modified (all are optional):

Name	Type	Description

modify-wsdl-address	boolean	This boolean enables and disables the address rewrite functionality. When modify-wsdl-address is set to true and the content of <soap:address> is a valid URL, JBossWS will rewrite the URL using the values of wsdl-host and wsdl-port or wsdl-secure-port. When modify-wsdl-address is set to false and the content of <soap:address> is a valid URL, JBossWS will not rewrite the URL. The <soap:address> URL will be used. When the content of <soap:address> is not a valid URL, JBossWS will rewrite it no matter what the setting of modify-wsdl-address. If modify-wsdl-address is set to true and wsdl-host is not defined or explicitly set to '`jbossws.undefined.host`' the content of <soap:address> URL is use. JBossWS uses the requester's host when rewriting the <soap:address> When modify-wsdl-address is not defined JBossWS uses a default value of true.
wsdl-host	string	The hostname / IP address to be used for rewriting `<soap:address>`. If `wsdl-host` is set to `jbossws.undefined.host`, JBossWS uses the requester's host when rewriting the `<soap:address>` When wsdl-host is not defined JBossWS uses a default value of '`jbossws.undefined.host`'.
wsdl-port	int	Set this property to explicitly define the HTTP port that will be used for rewriting the SOAP address. Otherwise the HTTP port will be identified by querying the list of installed HTTP connectors.
wsdl-secure-port	int	Set this property to explicitly define the HTTPS port that will be used for rewriting the SOAP address. Otherwise the HTTPS port will be identified by querying the list of installed HTTPS connectors.
wsdl-uri-scheme	string	This property explicitly sets the URI scheme to use for rewriting `<soap:address>` . Valid values are `http` and `https`. This configuration overrides scheme computed by processing the endpoint (even if a transport guarantee is specified). The provided values for `wsdl-port` and `wsdl-secure-port` (or their default values) are used depending on specified scheme.

wsdl-path-rewrite-rule	string	This string defines a SED substitution command (e.g., 's/regexp/replacement/g') that JBossWS executes against the path component of each <soap:address> URL published from the server. When wsdl-path-rewrite-rule is not defined, JBossWS retains the original path component of each <soap:address> URL. When 'modify-wsdl-address' is set to "false" this element is ignored.

Predefined endpoint configurations

JBossWS enables extra setup configuration data to be predefined and associated with an endpoint implementation. Predefined endpoint configurations can be used for JAX-WS client and JAX-WS endpoint setup. Endpoint configurations can include JAX-WS handlers and key/value properties declarations. This feature provides a convenient way to add handlers to WS endpoints and to set key/value properties that control JBossWS and Apache CXF internals (see Apache CXF configuration).

The webservices subsystem provides schema to support the definition of named sets of endpoint configuration data. Annotation, *org.jboss.ws.api.annotation.EndpointConfig* is provided to map the named configuration to the endpoint implementation.

There is no limit to the number of endpoint configurations that can be defined within the webservices subsystem. Each endpoint configuration must have a name that is unique within the webservices subsystem. Endpoint configurations defined in the webservices subsystem are available for reference by name through the annotation to any endpoint in a deployed application.

WildFly ships with two predefined endpoint configurations. Standard-Endpoint-Config is the default configuration. Recording-Endpoint-Config is an example of custom endpoint configuration and includes a recording handler.

```
[standalone@localhost:9999 /] /subsystem=webservices:read-resource
{
    "outcome" => "success",
    "result" => {
        "endpoint" => {},
        "modify-wsdl-address" => true,
        "wsdl-host" => expression "${jboss.bind.address:127.0.0.1}",
        "endpoint-config" => {
            "Standard-Endpoint-Config" => undefined,
            "Recording-Endpoint-Config" => undefined
        }
    }
}
```

The `Standard-Endpoint-Config` is a special endpoint configuration. It is used for any endpoint that does not have an explicitly assigned endpoint configuration.

Endpoint configs

Endpoint configs are defined using `the endpoint-config` element. Each endpoint configuration may include properties and handlers set to the endpoints associated to the configuration.

```
[standalone@localhost:9999 /]
/subsystem=webservices/endpoint-config=Recording-Endpoint-Config:read-resource
{
    "outcome" => "success",
    "result" => {
        "post-handler-chain" => undefined,
        "property" => undefined,
        "pre-handler-chain" => {"recording-handlers" => undefined}
    }
}
```

A new endpoint configuration can be added as follows:

```
[standalone@localhost:9999 /] /subsystem=webservices/endpoint-config=My-Endpoint-Config:add
{
    "outcome" => "success",
    "response-headers" => {
        "operation-requires-restart" => true,
        "process-state" => "restart-required"
    }
}
```

Handler chains

Each endpoint configuration may be associated with zero or more PRE and POST handler chains. Each handler chain may include JAXWS handlers. For outbound messages the PRE handler chains are executed before any handler that is attached to the endpoint using the standard means, such as with annotation @HandlerChain, and POST handler chains are executed after those objects have executed. For inbound messages the POST handler chains are executed before any handler that is attached to the endpoint using the standard means and the PRE handler chains are executed after those objects have executed.

```
* Server inbound messages
Client --> ... --> POST HANDLER --> ENDPOINT HANDLERS --> PRE HANDLERS --> Endpoint

* Server outbound messages
Endpoint --> PRE HANDLER --> ENDPOINT HANDLERS --> POST HANDLERS --> ... --> Client
```

The protocol-binding attribute must be used to set the protocols for which the chain will be triggered.

```
[standalone@localhost:9999 /]
/subsystem=webservices/endpoint-config=Recording-Endpoint-Config/pre-handler-chain=recording-handl
"outcome" => "success",
    "result" => {
        "protocol-bindings" => "##SOAP11_HTTP ##SOAP11_HTTP_MTOM ##SOAP12_HTTP
##SOAP12_HTTP_MTOM",
        "handler" => {"RecordingHandler" => undefined}
    },
    "response-headers" => {"process-state" => "restart-required"}
}
```

A new handler chain can be added as follows:

```
[standalone@localhost:9999 /]
/subsystem=webservices/endpoint-config=My-Endpoint-Config/post-handler-chain=my-handlers:add(proto
"outcome" => "success",
    "response-headers" => {
        "operation-requires-restart" => true,
        "process-state" => "restart-required"
    }
}
[standalone@localhost:9999 /]
/subsystem=webservices/endpoint-config=My-Endpoint-Config/post-handler-chain=my-handlers:read-reso
"outcome" => "success",
    "result" => {
        "handler" => undefined,
        "protocol-bindings" => "##SOAP11_HTTP"
    },
    "response-headers" => {"process-state" => "restart-required"}
}
```

Handlers

JAXWS handler can be added in handler chains:

```
[standalone@localhost:9999 /]
/subsystem=webservices/endpoint-config=Recording-Endpoint-Config/pre-handler-chain=recording-handl
"outcome" => "success",
    "result" => {"class" => "org.jboss.ws.common.invocation.RecordingServerHandler"},
    "response-headers" => {"process-state" => "restart-required"}
}
[standalone@localhost:9999 /]
/subsystem=webservices/endpoint-config=My-Endpoint-Config/post-handler-chain=my-handlers/handler=f
"outcome" => "success",
    "response-headers" => {
        "operation-requires-restart" => true,
        "process-state" => "restart-required"
    }
}
```

Endpoint-config handler classloading

The `class` attribute is used to provide the fully qualified class name of the handler. At deploy time, an instance of the class is created for each referencing deployment. For class creation to succeed, the deployment classloader must to be able to load the handler class.

Runtime information

Each web service endpoint is exposed through the deployment that provides the endpoint implementation. Each endpoint can be queried as a deployment resource. For further information please consult the chapter "Application Deployment". Each web service endpoint specifies a web context and a WSDL Url:

```
[standalone@localhost:9999 /] /deployment="*"/subsystem=webservices/endpoint="*":read-resource
{
   "outcome" => "success",
   "result" => [{
       "address" => [
           ("deployment" => "jaxws-samples-handlerchain.war"),
           ("subsystem" => "webservices"),
           ("endpoint" => "jaxws-samples-handlerchain:TestService")
       ],
       "outcome" => "success",
       "result" => {
           "class" => "org.jboss.test.ws.jaxws.samples.handlerchain.EndpointImpl",
           "context" => "jaxws-samples-handlerchain",
           "name" => "TestService",
           "type" => "JAXWS_JSE",
           "wsdl-url" => "http://localhost:8080/jaxws-samples-handlerchain?wsdl"
       }
   }]
}
```

Component Reference

The web service subsystem is provided by the JBossWS project. For a detailed description of the available configuration properties, please consult the project documentation.

- JBossWS homepage: http://www.jboss.org/jbossws
- Project Documentation: https://docs.jboss.org/author/display/JBWS

5.7.9 Resource adapters

Resource adapters are configured through the *resource-adapters* subsystem. Declaring a new resource adapter consists of two separate steps: You would need to deploy the .rar archive and define a resource adapter entry in the subsystem.

Resource Adapter Definitions

The resource adapter itself is defined within the subsystem *resource-adapters*:

```
<subsystem xmlns="urn:jboss:domain:resource-adapters:1.0">
    <resource-adapters>
       <resource-adapter>
          <archive>eis.rar</archive>
          <!-- Resource adapter level config-property -->
          <config-property name="Server">localhost</config-property>
          <config-property name="Port">19000</config-property>
          <transaction-support>XATransaction</transaction-support>
          <connection-definitions>
             <connection-definition class-name="com.acme.eis.ra.EISManagedConnectionFactory"
                                    jndi-name="java:/eis/AcmeConnectionFactory"
                                    pool-name="AcmeConnectionFactory">
                <!-- Managed connection factory level config-property -->
                <config-property name="Name">Acme Inc</config-property>
                <pool>
                   <min-pool-size>10</min-pool-size>
                   <max-pool-size>100</max-pool-size>
                </pool>
                <security>
                   <application/>
                </security>
             </connection-definition>
          </connection-definitions>
          <admin-objects>
              <admin-object class-name="com.acme.eis.ra.EISAdminObjectImpl"
                            jndi-name="java:/eis/AcmeAdminObject">
                 <config-property name="Threshold">10</config-property>
              </admin-object>
          </admin-objects>
       </resource-adapter>
    </resource-adapters>
</subsystem>
```

Note, that only JNDI bindings under java:/ or java:jboss/ are supported.

(See `standalone/configuration/standalone.xml`)

Using security domains

Information about using security domains can be found at
https://community.jboss.org/wiki/JBossAS7SecurityDomainModel

Automatic activation of resource adapter archives

A resource adapter archive can be automatically activated with a configuration by including an META-INF/ironjacamar.xml in the archive.

The schema can be found at http://docs.jboss.org/ironjacamar/schema/ironjacamar_1_0.xsd

Component Reference

The resource adapter subsystem is provided by the IronJacamar project. For a detailed description of the available configuration properties, please consult the project documentation.

- IronJacamar homepage: http://www.jboss.org/ironjacamar
- Project Documentation: http://www.jboss.org/ironjacamar/docs
- Schema description: http://docs.jboss.org/ironjacamar/userguide/1.0/en-US/html/deployment.html#deployingra_descriptor

5.7.10 Batch

Overview

The batch subsystem is used to configure an environment for running batch applications. WildFly uses JBeret for it's batch implementation. Specific information about JBeret can be found in the user guide. The resource path, in CLI notation, for the subsystem is `subsystem=batch-jberet`.

Default Subsystem Configuration

For up to date information about subsystem configuration options see http://wildscribe.github.io/.

Security

A new `security-domain` attribute was added to the `batch-jberet` subsystem to allow batch jobs to be executed under that security domain. Jobs that are stopped as part of a `suspend` operation will be restarted on execution of a `resume` with the original user that started job.

There was a `org.wildfly.extension.batch.jberet.deployment.BatchPermission` added to allow a security restraint to various batch functions. The following functions can be controlled with this permission.

- start
- stop
- restart
- abandon
- read

The read function allows users to use the getter methods from the `javax.batch.operations.JobOperator` or read the `batch-jberet` deployment resource, for example `/deployment=my.war/subsystem=batch-jberet:read-resource`.

Deployment Descriptors

There are no deployment descriptors for configuring a batch environment defined by the JSR-352 specification. In WildFly you can use a `jboss-all.xml` deployment descriptor to define aspects of the batch environment for your deployment.

In the `jboss-all.xml` deployment descriptor you can define a named job repository, a new job repository and/or a named thread pool. A named job repository and named thread pool are resources defined on the batch subsystem. Only a named thread pool is allowed to be defined in the deployment descriptor.

Example Named Job Repository and Thread Pool

```
<jboss umlns="urn:jboss:1.0">
    <batch xmlns="urn:jboss:batch-jberet:1.0">
      <job-repository>
        <named name="batch-ds"/>
      </job-repository>
      <thread-pool name="deployment-thread-pool"/>
    </batch>
</jboss>
```

Example new Job Repository

```
<jboss umlns="urn:jboss:1.0">
    <batch xmlns="urn:jboss:batch-jberet:1.0">
        <job-repository>
            <jdbc jndi-name="java:jboss/datasources/ExampleDS"/>
        </job-repository>
    </batch>
</jboss>
```

Deployment Resources

Some subsystems in WildFly register runtime resources for deployments. The batch subsystem registers jobs and executions. The jobs are registered using the job name, this is *not* the job XML name. Executions are registered using the execution id.

Batch application in a standalone server

```
[standalone@localhost:9990 /]
/deployment=batch-jdbc-chunk.war/subsystem=batch-jberet:read-resource(recursive=true,include-runti
"outcome" => "success",
    "result" => {"job" => {
        "reader-3" => {
            "instance-count" => 1,
            "running-executions" => 0,
            "execution" => {"1" => {
                "batch-status" => "COMPLETED",
                "create-time" => "2015-08-07T15:37:06.416-0700",
                "end-time" => "2015-08-07T15:37:06.519-0700",
                "exit-status" => "COMPLETED",
                "instance-id" => 1L,
                "last-updated-time" => "2015-08-07T15:37:06.519-0700",
                "start-time" => "2015-08-07T15:37:06.425-0700"
            }}
        },
        "reader-5" => {
            "instance-count" => 0,
            "running-executions" => 0,
            "execution" => undefined
        }
    }}
}
```

The batch subsystem resource on a deployment also has 3 operations to interact with batch jobs on the selected deployment. There is a `start-job`, `stop-job` and `restart-job` operation. The `execution` resource also has a `stop-job` and `restart-job` operation.

Example start-job

```
[standalone@localhost:9990 /]
/deployment=batch-chunk.war/subsystem=batch-jberet:start-job(job-xml-name=simple,
properties={writer.sleep=5000})
{
    "outcome" => "success",
    "result" => 1L
}
```

Example stop-job

```
[standalone@localhost:9990 /]
/deployment=batch-chunk.war/subsystem=batch-jberet:stop-job(execution-id=2)
```

Example restart-job

```
[standalone@localhost:9990 /]
/deployment=batch-chunk.war/subsystem=batch-jberet:restart-job(execution-id=2)
{
    "outcome" => "success",
    "result" => 3L
}
```

Result of resource after the 3 executions

```
[standalone@localhost:9990 /]
/deployment=batch-chunk.war/subsystem=batch-jberet:read-resource(recursive=true,
include-runtime=true)
{
    "outcome" => "success",
    "result" => {"job" => {"chunkPartition" => {
        "instance-count" => 2,
        "running-executions" => 0,
        "execution" => {
            "1" => {
                "batch-status" => "COMPLETED",
                "create-time" => "2015-08-07T15:41:55.504-0700",
                "end-time" => "2015-08-07T15:42:15.513-0700",
                "exit-status" => "COMPLETED",
                "instance-id" => 1L,
                "last-updated-time" => "2015-08-07T15:42:15.513-0700",
                "start-time" => "2015-08-07T15:41:55.504-0700"
            },
            "2" => {
                "batch-status" => "STOPPED",
                "create-time" => "2015-08-07T15:44:39.879-0700",
                "end-time" => "2015-08-07T15:44:54.882-0700",
                "exit-status" => "STOPPED",
                "instance-id" => 2L,
                "last-updated-time" => "2015-08-07T15:44:54.882-0700",
                "start-time" => "2015-08-07T15:44:39.879-0700"
            },
            "3" => {
                "batch-status" => "COMPLETED",
                "create-time" => "2015-08-07T15:45:48.162-0700",
                "end-time" => "2015-08-07T15:45:53.165-0700",
                "exit-status" => "COMPLETED",
                "instance-id" => 2L,
                "last-updated-time" => "2015-08-07T15:45:53.165-0700",
                "start-time" => "2015-08-07T15:45:48.163-0700"
            }
        }
    }}}
}
```

Pro Tip

You can filter jobs by an attribute on the execution resource with the `query` operation.

View all stopped jobs

```
/deployment=batch-chunk.war/subsystem=batch-jberet/job=*/execution=*:query(where=["batch-sta
"STOPPED"])
```

As with all operations you can see details about the operation using the `:read-operation-description` operation.

Tab completion

Don't forget that CLI has tab completion which will complete operations and attributes (arguments) on operations.

Example start-job operation description

```
[standalone@localhost:9990 /]
/deployment=batch-chunk.war/subsystem=batch-jberet:read-operation-description(name=start-job)
{
    "outcome" => "success",
    "result" => {
        "operation-name" => "start-job",
        "description" => "Starts a batch job.",
        "request-properties" => {
            "job-xml-name" => {
                "type" => STRING,
                "description" => "The name of the job XML file to use when starting the job.",
                "expressions-allowed" => false,
                "required" => true,
                "nillable" => false,
                "min-length" => 1L,
                "max-length" => 2147483647L
            },
            "properties" => {
                "type" => OBJECT,
                "description" => "Optional properties to use when starting the batch job.",
                "expressions-allowed" => false,
                "required" => false,
                "nillable" => true,
                "value-type" => STRING
            }
        },
        "reply-properties" => {"type" => LONG},
        "read-only" => false,
        "runtime-only" => true
    }
}
```

5.7.11 JSF

- Overview
- Installing a new JSF implementation manually
    - Add a module slot for the new JSF implementation JAR
    - Add a module slot for the new JSF API JAR
    - Add a module slot for the JSF injection JAR
    - For MyFaces only - add a module for the commons-digester JAR
    - Start the server
- Changing the default JSF implementation
- Configuring a JSF app to use a non-default JSF implementation

Overview

JSF configuration is handled by the JSF subsystem. The JSF subsystem allows multiple JSF implementations to be installed on the same WildFly server. In particular, any version of Mojarra or MyFaces that implements spec level 2.1 or higher can be installed. For each JSF implementation, a new slot needs to be created under `com.sun.jsf-impl`, `javax.faces.api`, and `org.jboss.as.jsf-injection`. When the JSF subsystem starts up, it scans the module path to find all of the JSF implementations that have been installed. The default JSF implementation that WildFly should use is defined by the `default-jsf-impl-slot` attribute.

Installing a new JSF implementation manually

A new JSF implementation can be manually installed as follows:

Add a module slot for the new JSF implementation JAR

- Create the following directory structure under the WILDFLY_HOME/modules directory:
  WILDFLY_HOME/modules/com/sun/jsf-impl/<JSF_IMPL_NAME>-<JSF_VERSION>

  For example, for Mojarra 2.2.11, the above path would resolve to:
  WILDFLY_HOME/modules/com/sun/jsf-impl/mojarra-2.2.11

- Place the JSF implementation JAR in the <JSF_IMPL_NAME>-<JSF_VERSION> subdirectory. In the same subdirectory, add a `module.xml` file similar to the Mojarra or MyFaces template examples. Change the `resource-root-path` to the name of your JSF implementation JAR and fill in appropriate values for ${`jsf-impl-name`} and ${`jsf-version`}.

Add a module slot for the new JSF API JAR

- Create the following directory structure under the WILDFLY_HOME/modules directory:
  WILDFLY_HOME/modules/javax/faces/api/<JSF_IMPL_NAME>-<JSF_VERSION>

- Place the JSF API JAR in the <JSF_IMPL_NAME>-<JSF_VERSION> subdirectory. In the same subdirectory, add a `module.xml` file similar to the Mojarra or MyFaces template examples. Change the `resource-root-path` to the name of your JSF API JAR and fill in appropriate values for ${`jsf-impl-name`} and ${`jsf-version`}.

Add a module slot for the JSF injection JAR

- Create the following directory structure under the WILDFLY_HOME/modules directory: WILDFLY_HOME/modules/org/jboss/as/jsf-injection/<JSF_IMPL_NAME>-<JSF_VERSION>
- Copy the wildfly-jsf-injection JAR and the weld-core-jsf JAR from WILDFLY_HOME/modules/system/layers/base/org/jboss/as/jsf-injection/main to the <JSF_IMPL_NAME>-<JSF_VERSION> subdirectory.
- In the <JSF_IMPL_NAME>-<JSF_VERSION> subdirectory, add a `module.xml` file similar to the Mojarra or MyFaces template examples and fill in appropriate values for ${`jsf-impl-name`}, ${`jsf-version`}, ${`version.jboss.as`}, and ${`version.weld.core`}. (These last two placeholders depend on the versions of the wildfly-jsf-injection and weld-core-jsf JARs that were copied over in the previous step.)

For MyFaces only - add a module for the commons-digester JAR

- Create the following directory structure under the WILDFLY_HOME/modules directory: WILDFLY_HOME/modules/org/apache/commons/digester/main
- Place the commons-digester JAR in WILDFLY_HOME/modules/org/apache/commons/digester/main. In the `main` subdirectory, add a `module.xml` file similar to this template. Fill in the appropriate value for ${`version.commons-digester`}.

Start the server

After starting the server, the following CLI command can be used to verify that your new JSF implementation has been installed successfully. The new JSF implementation should appear in the output of this command.

```
[standalone@localhost:9990 /] /subsystem=jsf:list-active-jsf-impls()
```

Changing the default JSF implementation

The following CLI command can be used to make a newly installed JSF implementation the default JSF implementation used by WildFly:

```
/subsystem=jsf:write-attribute(name=default-jsf-impl-slot,value=<JSF_IMPL_NAME>-<JSF_VERSION>)
```

A server restart will be required for this change to take effect.

Configuring a JSF app to use a non-default JSF implementation

A JSF app can be configured to use an installed JSF implementation that's not the default implementation by adding a `org.jboss.jbossfaces.JSF_CONFIG_NAME` context parameter to its `web.xml` file. For example, to indicate that a JSF app should use MyFaces 2.2.12 (assuming MyFaces 2.2.12 has been installed on the server), the following context parameter would need to be added:

```
<context-param>
  <param-name>org.jboss.jbossfaces.JSF_CONFIG_NAME</param-name>
  <param-value>myfaces-2.2.12</param-value>
</context-param>
```

If a JSF app does not specify this context parameter, the default JSF implementation will be used for that app.

5.7.12 JMX

The JMX subsystem registers a service with the Remoting endpoint so that remote access to JMX can be obtained over the exposed Remoting connector.

This is switched on by default in standalone mode and accessible over port 9990 but in domain mode is switched off so needs to be enabled - in domain mode the port will be the port of the Remoting connector for the WildFly instance to be monitored.

To use the connector you can access it in the standard way using a `service:jmx` URL:

```
import javax.management.MBeanServerConnection;
import javax.management.remote.JMXConnector;
import javax.management.remote.JMXConnectorFactory;
import javax.management.remote.JMXServiceURL;

public class JMXExample {

    public static void main(String[] args) throws Exception {
        //Get a connection to the WildFly MBean server on localhost
        String host = "localhost";
        int port = 9990;  // management-web port
        String urlString =
            System.getProperty("jmx.service.url","service:jmx:remote+http://" + host + ":" +
port);
        JMXServiceURL serviceURL = new JMXServiceURL(urlString);
        JMXConnector jmxConnector = JMXConnectorFactory.connect(serviceURL, null);
        MBeanServerConnection connection = jmxConnector.getMBeanServerConnection();

        //Invoke on the WildFly MBean server
        int count = connection.getMBeanCount();
        System.out.println(count);
        jmxConnector.close();
    }
}
```

You also need to set your classpath when running the above example. The following script covers Linux. If your environment is much different, paste your script when you have it working.

```
!/bin/bash

# specify your WildFly folder
export YOUR_JBOSS_HOME=~/WildFly

java -classpath $YOUR_JBOSS_HOME/bin/client/jboss-client.jar:./ JMXExample
```

You can also connect using jconsole.

If using jconsole use the `jconsole.sh` and `jconsole.bat` scripts included in the /bin directory of the WildFly distribution as these set the classpath as required to connect over Remoting.

In addition to the standard JVM MBeans, the WildFly MBean server contains the following MBeans:

JMX ObjectName	Description
`jboss.msc:type=container,name=jboss-as`	Exposes management operations on the JBoss Modular Service Container, which is the dependency injection framework at the heart of WildFly. It is useful for debugging dependency problems, for example if you are integrating your own subsystems, as it exposes operations to dump all services and their current states
`jboss.naming:type=JNDIView`	Shows what is bound in JNDI
`jboss.modules:type=ModuleLoader,name=*`	This collection of MBeans exposes management operations on JBoss Modules classloading layer. It is useful for debugging dependency problems arising from missing module dependencies

Audit logging

Audit logging for the JMX MBean server managed by the JMX subsystem. The resource is at `/subsystem=jmx/configuration=audit-log` and its attributes are similar to the ones mentioned for `/core-service=management/access=audit/logger=audit-log` in Audit logging.

Attribute	Description
`enabled`	`true` to enable logging of the JMX operations
`log-boot`	`true` to log the JMX operations when booting the server, `false` otherwise
`log-read-only`	If `true` all operations will be audit logged, if `false` only operations that change the model will be logged

Then which handlers are used to log the management operations are configured as `handler=*` children of the logger. These handlers and their formatters are defined in the global `/core-service=management/access=audit` section mentioned in Audit logging.

JSON Formatter

The same JSON Formatter is used as described in Audit logging. However the records for MBean Server invocations have slightly different fields from those logged for the core management layer.

```
2013-08-29 18:26:29 - {
    "type" : "jmx",
    "r/o" : false,
    "booting" : false,
    "version" : "10.0.0.Final",
    "user" : "$local",
    "domainUUID" : null,
    "access" : "JMX",
    "remote-address" : "127.0.0.1/127.0.0.1",
    "method" : "invoke",
    "sig" : [
        "javax.management.ObjectName",
        "java.lang.String",
        "[Ljava.lang.Object;",
        "[Ljava.lang.String;"
    ],
    "params" : [
        "java.lang:type=Threading",
        "getThreadInfo",
        "[Ljava.lang.Object;@5e6c33c",
        "[Ljava.lang.String;@4b681c69"
    ]
}
```

It includes an optional timestamp and then the following information in the json record

Field name	Description
`type`	This will have the value `jmx` meaning it comes from the jmx subsystem
`r/o`	`true` if the operation has read only impact on the MBean(s)
`booting`	`true` if the operation was executed during the bootup process, `false` if it was executed once the server is up and running
`version`	The version number of the WildFly instance
`user`	The username of the authenticated user.
`domainUUID`	This is not currently populated for JMX operations
`access`	This can have one of the following values: *`NATIVE` - The operation came in through the native management interface, for example the CLI *`HTTP` - The operation came in through the domain HTTP interface, for example the admin console *`JMX` - The operation came in through the JMX subsystem. See JMX for how to configure audit logging for JMX.
`remote-address`	The address of the client executing this operation
`method`	The name of the called MBeanServer method
`sig`	The signature of the called called MBeanServer method
`params`	The actual parameters passed in to the MBeanServer method, a simple `Object.toString()` is called on each parameter.
`error`	If calling the MBeanServer method resulted in an error, this field will be populated with `Throwable.getMessage()`

5.7.13 Deployment Scanner

The deployment scanner is only used in standalone mode. Its job is to monitor a directory for new files and to deploy those files. It can be found in `standalone.xml`:

```
<subsystem xmlns="urn:jboss:domain:deployment-scanner:2.0">
   <deployment-scanner scan-interval="5000"
      relative-to="jboss.server.base.dir" path="deployments" />
</subsystem>
```

You can define more `deployment-scanner` entries to scan for deployments from more locations. The configuration showed will scan the `JBOSS_HOME/standalone/deployments` directory every five seconds. The runtime model is shown below, and uses default values for attributes not specified in the xml:

```
[standalone@localhost:9999 /] /subsystem=deployment-scanner:read-resource(recursive=true)
{
    "outcome" => "success",
    "result" => {"scanner" => {"default" => {
        "auto-deploy-exploded" => false,
        "auto-deploy-zipped" => true,
        "deployment-timeout" => 60L,
        "name" => "default",
        "path" => "deployments",
        "relative-to" => "jboss.server.base.dir",
        "scan-enabled" => true,
        "scan-interval" => 5000
    }}}
}
```

The attributes are

Name	Type	Description
`name`	STRING	The name of the scanner. `default` is used if not specified
`path`	STRING	The actual filesystem path to be scanned. Treated as an absolute path, unless the 'relative-to' attribute is specified, in which case the value is treated as relative to that path.
`relative-to`	STRING	Reference to a filesystem path defined in the "paths" section of the server configuration, or one of the system properties specified on startup. In the example above `jboss.server.base.dir` resolves to `JBOSS_HOME/standalone`
`scan-enabled`	BOOLEAN	If true scanning is enabled
`scan-interval`	INT	Periodic interval, in milliseconds, at which the repository should be scanned for changes. A value of less than 1 indicates the repository should only be scanned at initial startup.
`auto-deploy-zipped`	BOOLEAN	Controls whether zipped deployment content should be automatically deployed by the scanner without requiring the user to add a .dodeploy marker file.
`auto-deploy-exploded`	BOOLEAN	Controls whether exploded deployment content should be automatically deployed by the scanner without requiring the user to add a .dodeploy marker file. Setting this to 'true' is not recommended for anything but basic development scenarios, as there is no way to ensure that deployment will not occur in the middle of changes to the content.
`auto-deploy-xml`	BOOLEAN	Controls whether XML content should be automatically deployed by the scanner without requiring a .dodeploy marker file.
`deployment-timeout`	LONG	Timeout, in seconds, a deployment is allows to execute before being canceled. The default is 60 seconds.

Deployment scanners can be added by modifying `standalone.xml` before starting up the server or they can be added and removed at runtime using the CLI

```
[standalone@localhost:9990 /]
/subsystem=deployment-scanner/scanner=new:add(scan-interval=10000,relative-to="jboss.server.base.d
=> "success"}
[standalone@localhost:9990 /] /subsystem=deployment-scanner/scanner=new:remove
{"outcome" => "success"}
```

You can also change the attributes at runtime, so for example to turn off scanning you can do

```
[standalone@localhost:9990 /]
/subsystem=deployment-scanner/scanner=default:write-attribute(name="scan-enabled",value=false)
{"outcome" => "success"}
[standalone@localhost:9990 /] /subsystem=deployment-scanner:read-resource(recursive=true)
{
    "outcome" => "success",
    "result" => {"scanner" => {"default" => {
        "auto-deploy-exploded" => false,
        "auto-deploy-zipped" => true,
        "deployment-timeout" => 60L,
        "name" => "default",
        "path" => "deployments",
        "relative-to" => "jboss.server.base.dir",
        "scan-enabled" => false,
        "scan-interval" => 5000
    }}}
}
```

5.7.14 Core Management

Overview

The core management subsystem is composed services used to manage the server or monitor its status. The core management subsystem configuration may be used to:

- register a listener for a server lifecycle events.
- list the last configuration changes on a server.

Lifecycle listener

You can create an implementation of *org.wildfly.extension.core.management.client.ProcessStateListener* which will be notified on running and runtime configuration state changes thus enabling the developer to react to those changes.

In order to use this feature you need to create your own module then configure and deploy it using the core management subsystem.

For example let's create a simple listener :

```
public class SimpleListener implements ProcessStateListener {

    private File file;
    private FileWriter fileWriter;
    private ProcessStateListenerInitParameters parameters;

    @Override
    public void init(ProcessStateListenerInitParameters parameters) {
        this.parameters = parameters;
        this.file = new File(parameters.getInitProperties().get("file"));
        try {
            fileWriter = new FileWriter(file, true);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void cleanup() {
        try {
            fileWriter.close();
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            fileWriter = null;
        }
    }

    @Override
    public void runtimeConfigurationStateChanged(RuntimeConfigurationStateChangeEvent evt) {
        try {
            fileWriter.write(String.format("%s %s %s %s\n", parameters.getProcessType(),
parameters.getRunningMode(), evt.getOldState(), evt.getNewState()));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void runningStateChanged(RunningStateChangeEvent evt) {
        try {
            fileWriter.write(String.format("%s %s %s %s\n", parameters.getProcessType(),
parameters.getRunningMode(), evt.getOldState(), evt.getNewState()));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

To compile it you need to depend on the *org.wildfly.core:wildfly-core-management-client* maven module.
Now let's add the module to the wildfly modules :

```
module add --name=org.simple.lifecycle.events.listener
--dependencies=org.wildfly.extension.core-management-client
--resources=/home/ehsavoie/dev/demo/simple-listener/target/simple-process-state-listener.jar
```

Now we can register or listener :

```
/subsystem=core-management/process-state-listener=simple-listener:add(class=org.simple.lifecycle.e
module=org.simple.lifecycle.events.listener, properties={file=/home/wildfly/tmp/events.txt})
```

Configuration changes

You can use the core management subsystem to enable and configure an **in-memory** history of the last configuration changes.
For example to track the last 5 configuration changes let's active this :

```
/subsystem=core-management/service=configuration-changes:add(max-history=5)
```

Now we can list the last configuration changes :

```
/subsystem=core-management/service=configuration-changes:list-changes()
{
    "outcome" => "success",
    "result" => [{
        "operation-date" => "2016-12-05T11:05:12.867Z",
        "access-mechanism" => "NATIVE",
        "remote-address" => "/127.0.0.1",
        "outcome" => "success",
        "operations" => [{
            "address" => [
                ("subsystem" => "core-management"),
                ("service" => "configuration-changes")
            ],
            "operation" => "add",
            "max-history" => 5,
            "operation-headers" => {
                "caller-type" => "user",
                "access-mechanism" => "NATIVE"
            }
        }]
    }]
}
```

5.7.15 Simple configuration subsystems

The following subsystems currently have no configuration beyond its root element in the configuration

```
<subsystem xmlns="urn:jboss:domain:jaxrs:1.0"/>
<subsystem xmlns="urn:jboss:domain:jdr:1.0"/>
<subsystem xmlns="urn:jboss:domain:pojo:1.0"/>
<subsystem xmlns="urn:jboss:domain:sar:1.0"/>
```

The presence of each of these turns on a piece of functionality:

Name	Description
`jaxrs`	Enables the deployment and functionality of JAX-RS applications
`jdr`	Enables the gathering of diagnostic data for use in remote analysis of error conditions. Although the data is in a simple format and could be useful to anyone, primarily useful for JBoss EAP subscribers who would provide the data to Red Hat when requesting support
`pojo`	Enables the deployment of applications containing JBoss Microcontainer services, as supported by previous versions of JBoss Application Server
`sar`	Enables the deployment of .SAR archives containing MBean services, as supported by previous versions of JBoss Application Server

5.8 Domain setup

To run a group of servers as a managed domain you need to configure both the domain controller and each host that joins the domain. This sections focuses on the network configuration for the domain and host controller components. For background information users are encouraged to review the Operating modes and Configuration Files sections.

5.8.1 Domain Controller Configuration

The domain controller is the central government for a managed domain. A domain controller configuration requires two steps:

- A host needs to be configured to act as the Domain Controller for the whole domain
- The host must expose an addressable management interface binding for the managed hosts to communicate with it

Example IP Addresses

In this example the domain controller uses 192.168.0.101 and the host controller 192.168.0.10

Configuring a host to act as the Domain Controller is done through the `domain-controller` declaration in `host.xml`. If it includes the `<local/>` element, then this host will become the domain controller:

```
<domain-controller>
   <local/>
</domain-controller>
```

(See domain/configuration/host.xml)

A host acting as the Domain Controller *must* expose a management interface on an address accessible to the other hosts in the domain. Exposing an HTTP(S) management interface is not required, but is recommended as it allows the Administration Console to work:

```
<management-interfaces>
  <native-interface security-realm="ManagementRealm">
    <socket interface="management" port="${jboss.management.native.port:9999}"/>
  </native-interface>
  <http-interface security-realm="ManagementRealm">
    <socket interface="management" port="${jboss.management.http.port:9990}"/>
  </http-interface>
</management-interfaces>
```

The interface attributes above refer to a named interface declaration later in the host.xml file. This interface declaration will be used to resolve a corresponding network interface.

```
<interfaces>
   <interface name="management">
       <inet-address value="192.168.0.101"/>
   </interface>
</interfaces>
```

(See domain/configuration/host.xml)

Please consult the chapter "Interface Configuration" for a more detailed explanation on how to configure network interfaces.

Next by default the master domain controller is configured to require authentication so a user needs to be added that can be used by the slave domain controller to connect.

Make use of the `add-user` utility to add a new user, for this example I am adding a new user called slave.

`add-user` MUST be run on the master domain controller and NOT the slave.

When you reach the final question of the interactive flow answer `y` or `yes` to indicate that the new user will be used for a process e.g.

```
Is this new user going to be used for one AS process to connect to another AS process e.g. slave
domain controller?
yes/no? y
To represent the user add the following to the server-identities definition <secret
value="cE3EBEkE=" />
```

Make a note of the XML Element output as that is going to be required within the slave configuration.

5.8.2 Host Controller Configuration

Once the domain controller is configured correctly you can proceed with any host that should join the domain. The host controller configuration requires three steps:

- The logical host name (within the domain) needs to be distinct
- The host controller needs to know the domain controller IP address

Provide a distinct, logical name for the host. In the following example we simply name it "slave":

```
<host xmlns="urn:jboss:domain:3.0"
     name="slave">
[...]
</host>
```

(See domain/configuration/host.xml)

If the `name` attribute is not set, the default name for the host will be the value of the `jboss.host.name` system property. If that is not set, the value of the `HOSTNAME` or `COMPUTERNAME` environment variable will be used, one of which will be set on most operating systems. If neither is set the name will be the value of `InetAddress.getLocalHost().getHostName()`.

A security realm needs to be defined to hold the identity of the slave. Since it is performing a specific purpose I would suggest a new realm is defined although it is possible to combine this with an existing realm.

```
<security-realm name="SlaveRealm">
  <server-identities>
    <secret value="cE3EBEkE=" />
  </server-identities>
</security-realm>
```

The `<secret />` element here is the one output from `add-user` previously. To create the `<secret />` element yourself the `value` needs to be the password encoded using Base64.

Tell it how to find the domain controller so it can register itself with the domain:

```
<domain-controller>
   <remote protocol="remote" host="192.168.0.101" port="9999" username="slave"
security-realm="SlaveRealm"/>
</domain-controller>
```

Since we have also exposed the HTTP management interface we could also use :

```
<domain-controller>
   <remote protocol="http-remoting" host="192.168.0.101" port="9990" username="slave"
security-realm="SlaveRealm"/>
</domain-controller>
```

(See domain/configuration/host.xml)

The username attribute here is optional, if it is omitted then the name of the host will be used instead, in this example that was already set to name.

The name of each host needs to be unique when registering with the domain controller, however the username does not - using the username attribute allows the same account to be used by multiple hosts if this makes sense in your environment.

The `<remote />` element is also associated with the security realm `SlaveRealm`, this is how it picks up the password from the `<secret />` element.

Ignoring domain wide resources

WildFly 10 and later make it easy for slave host controllers to "ignore" parts of the domain wide configuration. What does the mean and why is it useful?

One of the responsibilities of the Domain Controller is ensuring that all running Host Controllers have a consistent local copy of the domain wide configuration (i.e. those resources whose address does not begin with `/host=*`, i.e. those that are persisted in `domain.xml`. Having that local copy allows a user to do the following things:

- Ask the slave to launch its already configured servers, even if the Domain Controller is not running.
- Configured new servers, using different server groups from those current running, and ask the slave to launch them, even if the Domain Controller is not running.
- Reconfigure the slave to act as the Domain Controller, allowing it to take over as the master if the previous master has failed or been shut down.

However, of these three things only the latter two require that the slave maintain a *complete* copy of the domain wide configuration. The first only requires the slave to have the *portion* of the domain wide configuration that is relevant to its current servers. And the first use case is the most common one. A slave that is only meant to support the first use case can safely "ignore" portions of the domain wide configuration. And there are benefits to ignoring some resources:

- If a server group is ignored, and the deployments mapped to that server group aren't mapped to other non-ignored groups, then the slave does not need to pull down a copy of the deployment content from the master. That can save disk space on the slave, improve the speed of starting new hosts and reduce network traffic.
- WildFly supports "mixed domains" where a later version Domain Controller can manage slaves running previous versions. But those "legacy" slaves cannot understand configuration resources, attributes and operations introduced in newer versions. So any attempt to use newer things in the domain wide configuration will fail unless the legacy slaves are ignoring the relevant resources. But ignoring resources will allow the legacy slaves to work fine managing servers using profiles without new concepts, while other hosts can run servers with profiles that take advantage of the latest features.

Prior to WildFly 10, a slave could be configured to ignore some resources, but the mechanism was not particularly user friendly:

- The resources to be ignored had to be listed in a fair amount of detail in each host's configuration.
- If a new resource is added and needs to be ignored, then **each** host that needs to ignore that must be updated to record that.

Starting with WildFly 10, this kind of detailed configuration is no longer required. Instead, with the standard versions of `host.xml`, the slave will behave as follows:

- If the slave was started with the `--backup` command line parameter, the behavior will be the same as releases prior to 10; i.e. only resources specifically configured to be ignored will be ignored.
- Otherwise, the slave will "ignore unused resources".

What does "ignoring unused resources" mean?

- Any server-group that is not referenced by one of the host's server-config resources is ignored.
- Any profile that is not referenced by a non-ignored server-group, either directly or indirectly via the profile resource's 'include' attribute, is ignored
- Any socket-binding-group that is not directly referenced by one of the host's server-config resources, or referenced by a non-ignored server-group, is ignored
- Extension resources will not be automatically ignored, even if no non-ignored profile uses the extension. Ignoring an extension requires explicit configuration. Perhaps in a future release extensions will be explicitly ignored.
- If a change is made to the slave host's configuration or to the domain wide configuration that reduces the set of ignored resources, then as part of handling that change the slave will contact the master to pull down the missing pieces of configuration and will integrate those pieces in its local copy of the management model. Examples of such changes include adding a new server-config that references a previously ignored server-group or socket-binding-group, changing the server-group or socket-binding-group assigned to a server-config, changing the profile or socket-binding-group assigned to a non-ignored server-group, or adding a profile or socket-binding-group to the set of those included directly or indirectly by a non-ignored profile or socket-binding-group.

The default behavior can be changed, either to always ignore unused resources, even if `--backup` is used, or to not ignore unused resources, by updating the domain-controller element in the `host-xml` file and setting the `ignore-unused-configuration` attribute:

```
<domain-controller>
        <remote security-realm="ManagementRealm" ignore-unused-configuration="false">
            <discovery-options>
                <static-discovery name="primary"
protocol="${jboss.domain.master.protocol:remote}" host="${jboss.domain.master.address}"
port="${jboss.domain.master.port:9999}"/>
            </discovery-options>
        </remote>
    </domain-controller>
```

The "ignore unused resources" behavior can be used in combination with the pre-WildFly 10 detailed specification of what to ignore. If that is done both the unused resources and the explicitly declared resources will be ignored. Here's an example of such a configuration, one where the slave cannot use the "org.example.foo" extension that has been installed on the Domain Controller and on some slaves, but not this one:

```
<domain-controller>
        <remote security-realm="ManagementRealm" ignore-unused-configuration="true">
            <ignored-resources type="extension">
                <instance name="org.example.foo"/>
            </ignored-resources>
            <discovery-options>
                <static-discovery name="primary"
protocol="${jboss.domain.master.protocol:remote}" host="${jboss.domain.master.address}"
port="${jboss.domain.master.port:9999}"/>
            </discovery-options>
        </remote>
    </domain-controller>
```

5.8.3 Server groups

The domain controller defines one or more server groups and associates each of these with a profile and a socket binding group, and also :

```
<server-groups>
    <server-group name="main-server-group" profile="default">
        <jvm name="default">
           <heap size="64m" max-size="512m"/>
           <permgen size="128m" max-size="128m"/>
        </jvm>
        <socket-binding-group ref="standard-sockets"/>
    </server-group>
    <server-group name="other-server-group" profile="bigger">
        <jvm name="default">
            <heap size="64m" max-size="512m"/>
        </jvm>
        <socket-binding-group ref="bigger-sockets"/>
    </server-group>
</server-groups>
```

(See domain/configuration/domain.xml)

The domain controller also defines the socket binding groups and the profiles. The socket binding groups define the default socket bindings that are used:

```
<socket-binding-groups>
    <socket-binding-group name="standard-sockets" default-interface="public">
        <socket-binding name="http" port="8080"/>
        [...]
    </socket-binding-group>
    <socket-binding-group name="bigger-sockets" include="standard-sockets"
default-interface="public">
        <socket-binding name="unique-to-bigger" port="8123"/>
    </socket-binding-group>
</socket-binding-groups>
```

(See domain/configuration/domain.xml)

In this example the `bigger-sockets` group includes all the socket bindings defined in the `standard-sockets` groups and then defines an extra socket binding of its own.

A profile is a collection of subsystems, and these subsystems are what implement the functionality people expect of an application server.

```
<profiles>
    <profile name="default">
        <subsystem xmlns="urn:jboss:domain:web:1.0">
            <connector name="http" scheme="http" protocol="HTTP/1.1" socket-binding="http"/>
            [...]
        </subsystem>
        <\!-\- The rest of the subsystems here \-->
        [...]
    </profile>
    <profile name="bigger">
        <subsystem xmlns="urn:jboss:domain:web:1.0">
            <connector name="http" scheme="http" protocol="HTTP/1.1" socket-binding="http"/>
            [...]
        </subsystem>
        <\!-\- The same subsystems as defined by 'default' here \-->
        [...]
        <subsystem xmlns="urn:jboss:domain:fictional-example:1.0">
            <socket-to-use name="unique-to-bigger"/>
        </subsystem>
    </profile>
</profiles>
```

(See domain/configuration/domain.xml)

Here we have two profiles. The `bigger` profile contains all the same subsystems as the `default` profile (athough the parameters for the various subsystems could be different in each profile), and adds the `fictional-example` subsystem which references the `unique-to-bigger` socket binding.

5.8.4 Servers

The host controller defines one or more servers:

```
<servers>
    <server name="server-one" group="main-server-group">
        <\!-\- server-one inherits the default socket-group declared in the server-group \-->
        <jvm name="default"/>
    </server>

    <server name="server-two" group="main-server-group" auto-start="true">
        <socket-binding-group ref="standard-sockets" port-offset="150"/>
        <jvm name="default">
            <heap size="64m" max-size="256m"/>
        </jvm>
    </server>

    <server name="server-three" group="other-server-group" auto-start="false">
        <socket-binding-group ref="bigger-sockets" port-offset="250"/>
    </server>
</servers>
```

(See domain/configuration/host.xml)

`server-one` and `server-two` both are associated with `main-server-group` so that means they both run the subsystems defined by the `default` profile, and have the socket bindings defined by the `standard-sockets` socket binding group. Since all the servers defined by a host will be run on the same physical host we would get port conflicts unless we used `<socket-binding-group ref="standard-sockets" port-offset="150"/>` for `server-two`. This means that `server-two` will use the socket bindings defined by `standard-sockets` but it will add `150` to each port number defined, so the value used for `http` will be `8230` for `server-two`.

`server-three` will not be started due to its `auto-start="false"`. The default value if no `auto-start` is given is `true` so both `server-one` and `server-two` will be started when the host controller is started. `server-three` belongs to `other-server-group`, so if its `auto-start` were changed to `true` it would start up using the subsystems from the `bigger` profile, and it would use the `bigger-sockets` socket binding group.

JVM

The host controller contains the main `jvm` definitions with arguments:

```
<jvms>
    <jvm name="default">
        <heap size="64m" max-size="128m"/>
    </jvm>
</jvms>
```

(See domain/configuration/host.xml)
From the preceeding examples we can see that we also had a `jvm` reference at server group level in the domain controller. The jvm's name **must** match one of the definitions in the host controller. The values supplied at domain controller and host controller level are combined, with the host controller taking precedence if the same parameter is given in both places.

Finally, as seen, we can also override the `jvm` at server level. Again, the jvm's name **must** match one of the definitions in the host controller. The values are combined with the ones coming in from domain controller and host controller level, this time the server definition takes precedence if the same parameter is given in all places.

Following these rules the jvm parameters to start each server would be

Server	JVM parameters
`server-one`	`-Xms64m -Xmx128m`
`server-two`	`-Xms64m -Xmx256m`
`server-three`	`-Xms64m -Xmx128m`

5.9 Other management tasks

5.9.1 Controlling operation via command line parameters

To start up a WildFly managed domain, execute the `$JBOSS_HOME/bin/domain.sh` script. To start up a standalone server, execute the `$JBOSS_HOME/bin/standalone.sh`. With no arguments, the default configuration is used. You can override the default configuration by providing arguments on the command line, or in your calling script.

System properties

To set a system property, pass its new value using the standard jvm `-Dkey=value` options:

```
$JBOSS_HOME/bin/standalone.sh -Djboss.home.dir=some/location/wildFly \
    -Djboss.server.config.dir=some/location/wildFly/custom-standalone
```

This command starts up a standalone server instance using a non-standard AS home directory and a custom configuration directory. For specific information about system properties, refer to the definitions below.

Instead of passing the parameters directly, you can put them into a properties file, and pass the properties file to the script, as in the two examples below.

```
$JBOSS_HOME/bin/domain.sh --properties=/some/location/jboss.properties
$JBOSS_HOME/bin/domain.sh -P=/some/location/jboss.properties
```

Note however, that properties set this way are not processed as part of JVM launch. They are processed early in the boot process, but this mechanism should not be used for setting properties that control JVM behavior (e.g. java.net.perferIPv4Stack) or the behavior of the JBoss Modules classloading system.

The syntax for passing in parameters and properties files is the same regardless of whether you are running the `domain.sh`, `standalone.sh`, or the Microsoft Windows scripts`domain.bat` or `standalone.bat`.

The properties file is a standard Java property file containing `key=value` pairs:

```
jboss.home.dir=/some/location/wildFly
jboss.domain.config.dir=/some/location/wildFly/custom-domain
```

System properties can also be set via the xml configuration files. Note however that for a standalone server properties set this way will not be set until the xml configuration is parsed and the commands created by the parser have been executed. So this mechanism should not be used for setting properties whose value needs to be set before this point.

Controlling filesystem locations with system properties

The standalone and the managed domain modes each use a default configuration which expects various files and writable directories to exist in standard locations. Each of these standard locations is associated with a system property, which has a default value. To override a system property, pass its new value using the one of the mechanisms above. The locations which can be controlled via system property are:

Standalone

Property name	Usage	Default value
`java.ext.dirs`	The JDK extension directory paths	`null`
`jboss.home.dir`	The root directory of the WildFly installation.	Set by `standalone.sh` to `$JBOSS_HOME`
`jboss.server.base.dir`	The base directory for server content.	`jboss.home.dir`/standalone
`jboss.server.config.dir`	The base configuration directory.	`jboss.server.base.dir` /configuration
`jboss.server.data.dir`	The directory used for persistent data file storage.	`jboss.server.base.dir` /data
`jboss.server.log.dir`	The directory containing the `server.log` file.	`jboss.server.base.dir`/log
`jboss.server.temp.dir`	The directory used for temporary file storage.	`jboss.server.base.dir`/tmp
`jboss.server.deploy.dir`	The directory used to store deployed content	`jboss.server.data.dir` /content

Managed Domain

Property name	Usage	Default value
`jboss.home.dir`	The root directory of the WildFly installation.	Set by `domain.sh` to `$JBOSS_HOME`
`jboss.domain.base.dir`	The base directory for domain content.	`jboss.home.dir`/domain
`jboss.domain.config.dir`	The base configuration directory	`jboss.domain.base.dir`/configuration
`jboss.domain.data.dir`	The directory used for persistent data file storage.	`jboss.domain.base.dir`/data
`jboss.domain.log.dir`	The directory containing the `host-controller.log` and `process-controller.log` files	`jboss.domain.base.dir`/log
`jboss.domain.temp.dir`	The directory used for temporary file storage	`jboss.domain.base.dir`/tmp
`jboss.domain.deployment.dir`	The directory used to store deployed content	`jboss.domain.base.dir`/content
`jboss.domain.servers.dir`	The directory containing the output for the managed server instances	`jboss.domain.base.dir`/servers

Other command line parameters

The first acceptable format for command line arguments to the WildFly launch scripts is

```
--name=value
```

For example:

```
$JBOSS_HOME/bin/standalone.sh --server-config=standalone-ha.xml
```

If the parameter name is a single character, it is prefixed by a single '-' instead of two. Some parameters have both a long and short option.

```
-x=value
```

For example:

```
$JBOSS_HOME/bin/standalone.sh -P=/some/location/jboss.properties
```

For some command line arguments frequently used in previous major releases of WildFly, replacing the "=" in the above examples with a space is supported, for compatibility.

```
-b 192.168.100.10
```

If possible, use the `-x=value` syntax. New parameters will always support this syntax.

The sections below describe the command line parameter names that are available in standalone and domain mode.

Standalone

Name	Default if absent	Value
`--admin-only`	-	Set the server's running type to ADMIN_ONLY causing it to open administrative interfaces and accept management requests but not start other runtime services or accept end user requests.
`--server-config` `-c`	standalone.xml	A relative path which is interpreted to be relative to `jboss.server.config.dir`. The name of the configuration file to use.
`--read-only-server-config`	-	A relative path which is interpreted to be relative to `jboss.server.config.dir`. This is similar to `--server-config` but if this alternative is specified the server will not overwrite the file when the management model is changed. However a full versioned history is maintained of the file.

Managed Domain

Name	Default if absent	Value
`--admin-only`	-	Set the server's running type to ADMIN_ONLY causing it to open administrative interfaces and accept management requests but not start servers or, if this host controller is the master for the domain, accept incoming connections from slave host controllers.
`--domain-config` `-c`	domain.xml	A relative path which is interpreted to be relative to `jboss.domain.config.dir`. The name of the domain wide configuration file to use.
`--read-only-domain-config`	-	A relative path which is interpreted to be relative to `jboss.domain.config.dir`. This is similar to `--domain-config` but if this alternative is specified the host controller will not overwrite the file when the management model is changed. However a full versioned history is maintained of the file.
`--host-config`	host.xml	A relative path which is interpreted to be relative to `jboss.domain.config.dir`. The name of the host-specific configuration file to use.
`--read-only-host-config`	-	A relative path which is interpreted to be relative to `jboss.domain.config.dir`. This is similar to `--host-config` but if this alternative is specified the host controller will not overwrite the file when the management model is changed. However a full versioned history is maintained of the file.

The following parameters take no value and are only usable on slave host controllers (i.e. hosts configured to connect to a `remote` domain controller.)

Name	Function
`--backup`	Causes the slave host controller to create and maintain a local copy (domain.cached-remote.xml) of the domain configuration. If *ignore-unused-configuration* is unset in host.xml, a complete copy of the domain configuration will be stored locally, otherwise the configured value of *ignore-unused-configuration* in host.xml will be used. (See ignore-unused-configuration for more details.)
`--cached-dc`	If the slave host controller is unable to contact the master domain controller to get its configuration at boot, this option will allow the slave host controller to boot and become operational using a previously cached copy of the domain configuration (domain.cached-remote.xml.) If the cached configuration is not present, this boot will fail. This file is created using using one of the following methods: - A previously successful connection to the master domain controller using --backup or --cached-dc. - Copying the domain configuration from an alternative host to domain/configuration/domain.cached-remote.xml. The unavailable master domain controller will be polled periodically for availability, and once becoming available, the slave host controller will reconnect to the master host controller and synchronize the domain configuration. During the interval the master domain controller is unavailable, the slave host controller will not be able make any modifications to the domain configuration, but it may launch servers and handle requests to deployed applications etc.

Common parameters

These parameters apply in both standalone or managed domain mode:

Name	Function
`-b=<value>`	Sets system property `jboss.bind.address` to <value>. See Controlling the Bind Address with -b for further details.
`-b<name>=<value>`	Sets system property `jboss.bind.address.`<name> to <value> where *name* can vary. See Controlling the Bind Address with -b for further details.
`-u=<value>`	Sets system property `jboss.default.multicast.address` to <value>. See Controlling the Default Multicast Address with -u for further details.

`--version`
`-v`
`-V`
	Prints the version of WildFly to standard output and exits the JVM.

`--help` `-h`	Prints a help message explaining the options and exits the JVM.

Controlling the Bind Address with -b

WildFly binds sockets to the IP addresses and interfaces contained in the `<interfaces>` elements in `standalone.xml`, `domain.xml` and `host.xml`. (See Interfaces and Socket Bindings for further information on these elements.) The standard configurations that ship with WildFly includes two interface configurations:

```
<interfaces>
    <interface name="management">
        <inet-address value="${jboss.bind.address.management:127.0.0.1}"/>
    </interface>
    <interface name="public">
       <inet-address value="${jboss.bind.address:127.0.0.1}"/>
    </interface>
</interfaces>
```

Those configurations use the values of system properties `jboss.bind.address.management` and `jboss.bind.address` if they are set. If they are not set, 127.0.0.1 is used for each value.

As noted in Common Parameters, the AS supports the `-b` and `-b<name>` command line switches. The only function of these switches is to set system properties `jboss.bind.address` and `jboss.bind.address.`<name> respectively. However, because of the way the standard WildFly configuration files are set up, using the `-b` switches can indirectly control how the AS binds sockets.

If your interface configurations match those shown above, using this as your launch command causes all sockets associated with interface named "public" to be bound to `192.168.100.10`.

```
$JBOSS_HOME/bin/standalone.sh -b=192.168.100.10
```

In the standard config files, public interfaces are those not associated with server management. Public interfaces handle normal end-user requests.

Interface names

The interface named "public" is not inherently special. It is provided as a convenience. You can name your interfaces to suit your environment.

To bind the public interfaces to all IPv4 addresses (the IPv4 wildcard address), use the following syntax:

```
$JBOSS_HOME/bin/standalone.sh -b=0.0.0.0
```

You can also bind the management interfaces, as follows:

```
$JBOSS_HOME/bin/standalone.sh -bmanagement=192.168.100.10
```

In the standard config files, management interfaces are those sockets associated with server management, such as the socket used by the CLI, the HTTP socket used by the admin console, and the JMX connector socket.

Be Careful

The `-b` switch only controls the interface bindings because the standard config files that ship with WildFly sets things up that way. If you change the `<interfaces>` section in your configuration to no longer use the system properties controlled by `-b`, then setting `-b` in your launch command will have no effect.

For example, this perfectly valid setting for the "public" interface causes `-b` to have no effect on the "public" interface:

```
<interface name="public">
   <nic name="eth0"/>
</interface>
```

The key point is **the contents of the configuration files determine the configuration. Settings like `-b` are not overrides of the configuration files.** They only provide a shorter syntax for setting a system properties that may or may not be referenced in the configuration files. They are provided as a convenience, and you can choose to modify your configuration to ignore them.

Controlling the Default Multicast Address with -u

WildFly may use multicast communication for some services, particularly those involving high availability clustering. The multicast addresses and ports used are configured using the `socket-binding` elements in `standalone.xml` and `domain.xml`. (See Socket Bindings for further information on these elements.) The standard HA configurations that ship with WildFly include two socket binding configurations that use a default multicast address:

```
<socket-binding name="jgroups-mping" port="0"
multicast-address="${jboss.default.multicast.address:230.0.0.4}" multicast-port="45700"/>
<socket-binding name="jgroups-udp" port="55200"
multicast-address="${jboss.default.multicast.address:230.0.0.4}" multicast-port="45688"/>
```

Those configurations use the values of system property `jboss.default.multicast.address` if it is set. If it is not set, 230.0.0.4 is used for each value. (The configuration may include other socket bindings for multicast-based services that are not meant to use the default multicast address; e.g. a binding the mod-cluster services use to communicate on a separate address/port with Apache httpd servers.)

As noted in Common Parameters, the AS supports the `-u` command line switch. The only function of this switch is to set system property `jboss.default.multicast.address`. However, because of the way the standard AS configuration files are set up, using the `-u` switches can indirectly control how the AS uses multicast.

If your socket binding configurations match those shown above, using this as your launch command causes the service using those sockets configurations to be communicate over multicast address `230.0.1.2`.

```
$JBOSS_HOME/bin/standalone.sh -u=230.0.1.2
```

Be Careful

As with the `-b` switch, the `-u` switch only controls the multicast address used because the standard config files that ship with WildFly sets things up that way. If you change the `<socket-binding>` sections in your configuration to no longer use the system properties controlled by `-u`, then setting `-u` in your launch command will have no effect.

5.9.2 Suspend, resume and graceful shutdown

Core Concepts

Wildfly introduces the ability to suspend and resume servers. This can be combined with shutdown to enable the server to gracefully finish processing all active requests and then shut down. When a server is suspended it will immediately stop accepting new requests, but wait for existing request to complete. A suspended server can be resumed at any point, and will begin processing requests immediately. Suspending and resuming has no effect on deployment state (e.g. if a server is suspended singleton EJB's will not be destroyed). As of Wildfly 11 it is also possible to start a server in suspended mode which means it will not accept requests until it has been resumed, servers will also be suspended during the boot process, so no requests will be accepted until the startup process is 100% complete.

Suspend/Resume has no effect on management operations, management operations can still be performed while a server is suspended. If you wish to perform a management operation that will affect the operation of the server (e.g. changing a datasource) you can suspend the server, perform the operation, then resume the server. This allows all requests to finish, and makes sure that no requests are running while the management changes are taking place.

When a server is suspending it goes through four different phases:

- **RUNNING** - The normal state, the server is accepting requests and running normally
- **PRE_SUSPEND** - In PRE_SUSPEND the server will notify external parties that it is about to suspend, for example mod_cluster will notify the load balancer that the deployment is suspending. Requests are still accepted in this phase.
- **SUSPENDING** - All new requests are rejected, and the server is waiting for all active requests to finish. If there are no active requests at suspend time this phase will be skipped.
- **SUSPENDED** - All requests have completed, and the server is suspended.

Starting Suspended

In order to start into suspended mode when using a standalone server you need to add **--start-mode=suspend** to the command line. It is also possible to specify the start-mode in the **reload** operation to cause the server to reload into suspended mode (other possible values for start-mode are **normal** and **admin-only**).

In domain mode servers can be started in suspended mode by passing the **suspend=true** parameter to any command that causes a server to start, restart or reload (e.g. :start-servers(suspend=true)).

The Request Controller Subsystem

Wildfly introduces a new subsystem called the Request Controller Subsystem. This optional subsystem tracks all requests at their entry point, which how the graceful shutdown mechanism know when all requests are done (it also allows you to provide a global limit on the total number of running requests).

If this subsystem is not present suspend/resume will be limited, in general things that happen in the PRE_SUSPEND phase will work as normal (stopping message delivery, notifying the load balancer), however the server will not wait for all requests to complete and instead move straight to SUSPENDED mode.

There is a small performance penalty associated with the request controller subsystem (about on par with enabling statistics), so if you do not require the suspend/resume functionality this subsystem can be removed to get a small performance boost.

Subsystem Integrations

Suspend/Resume is a service provided by the Wildfly platform that any subsystem may choose to integrate with. Some subsystems integrate directly with the suspend controller, while others integrate through the request controller subsystem.

The following subsystems support graceful shutdown. Note that only subsystems that provide an external entry point to the server need graceful shutdown support, for example the JAX-RS subsystem does not require suspend/resume support as all access to JAX-RS is through the web connector.

- **Undertow** - Undertow will wait for all requests to finish
- **mod_cluster** - The mod_cluster subsystem will notify the load balancer that the server is suspending in the PRE_SUSPEND phase.
- **EJB** - EJB will wait for all remote EJB requests and MDB message deliveries to finish. Delivery to MDB's is stopped in the PRE_SUSPEND phase. EJB timers are suspended, and missed timers will be activated when the server is resumed.
- **Batch** - Batch jobs will be stopped at a checkpoint while the server is suspending. They will be restarted from that checkpoint when the server returns to running mode.
- **EE Concurrency** - The server will wait for all active jobs to finish. All jobs that have already been queued will be skipped.
- **Transactions** - transaction subsystem waits for all running transactions to finish while server is suspending. During that time server refuses to start any new transaction. But any in-flight transaction will be serviced - e.g. it means that server accepts any incoming remote call which carries context of the transaction already started at the suspending server.
  When you work with EJBs you have to enable the graceful shutdown functionality by setting attribute `enable-graceful-txn-shutdown` to `true`.
  (at the `ejb3 subsystem` xml, for example):
  `<enable-graceful-txn-shutdown value="false"/>`
  By **default** graceful shutdown it's **disabled** for ejb subsystem.
  The reason is that the behavior might be unwelcome in cluster environments, as the server notifies remote clients that the node is no longer available for remote calls only after the transactions are finished. During that brief window of time, the client of a cluster may send a new request to a node that is shutting down and will refuse the request because it is not related to an existing transaction.
  If this attribute `enable-graceful-txn-shutdown` is set to `false`, we disable the graceful behavior and EJB clients will not attempt to invoke the node when it suspends, regardless of active transactions.

Standalone Mode

Suspend/Resume can be controlled via the following CLI operations in standalone mode:

```
:suspend(timeout=z)
```

Suspends the server. If the timeout is specified it will wait up to the specified number of seconds for all requests to finish. If there is no timeout specified or the value is less than zero it will wait indefinitely.

```
:resume
```

Resumes a previously suspended server. The server should be able to begin serving requests immediately.

```
:read-attribute(name=suspend-state)
```

Returns the current suspend state of the server.

```
:shutdown(timeout=x)
```

If a timeout parameter is passed to the shutdown command then a graceful shutdown will be performed. The server will be suspended, and will wait up to the specified number of seconds for all requests to finish before shutting down. A timeout value of less than zero means it will wait indefinitely.

Domain Mode

Domain mode has similar commands as standalone mode, however they can be applied at both the global and server group levels:

Whole Domain

```
:suspend-servers(timeout=x)
```

:resume-servers

:stop-servers(timeout=x)

Server Group

```
/server-group=main-server-group:suspend-servers(timeout=x)
```

```
/server-group=main-server-group:resume-servers
```

```
/server-group=main-server-group:stop-servers(timeout=x)
```

Server

```
/host=master/server-config=server-one:suspend(timeout=x)
```

/host=master/server-config=server-one:resume

/host=master/server-config=server-one:stop(timeout=x)

5.9.3 Starting & stopping Servers in a Managed Domain

Starting a standalone server is done through the `bin/standalone.sh` script. However in a managed domain server instances are managed by the domain controller and need to be started through the management layer:

First of all, get to know which `servers` are configured on a particular `host`:

```
[domain@localhost:9990 /] :read-children-names(child-type=host)
{
   "outcome" => "success",
   "result" => ["local"]
}


[domain@localhost:9990 /] /host=local:read-children-names(child-type=server-config)
{
   "outcome" => "success",
   "result" => [
       "my-server",
       "server-one",
       "server-three"
   ]
}
```

Now that we know, that there are two `servers` configured on `host` "*local*", we can go ahead and check their status:

```
[domain@localhost:9990 /]
/host=local/server-config=server-one:read-resource(include-runtime=true)
{
   "outcome" => "success",
   "result" => {
       "auto-start" => true,
       "group" => "main-server-group",
       "interface" => undefined,
       "name" => "server-one",
       "path" => undefined,
       "socket-binding-group" => undefined,
       "socket-binding-port-offset" => undefined,
       "status" => "STARTED",
       "system-property" => undefined,
       "jvm" => {"default" => undefined}
   }
}
```

You can change the server state through the "*start*" and "*stop*" operations

```
[domain@localhost:9990 /] /host=local/server-config=server-one:stop
{
   "outcome" => "success",
   "result" => "STOPPING"
}
```

Navigating through the domain topology is much more simple when you use the web interface.

5.9.4 Controlling JVM settings

Configuration of the JVM settings is different for a managed domain and a standalone server. In a managed domain, the domain controller components are responsible for starting and stoping server processes and hence determine the JVM settings. For a standalone server, it's the responsibility of the process that started the server (e.g. passing them as command line arguments).

Managed Domain

In a managed domain the JVM settings can be declared at different scopes: For a specific server group, for a host or for a particular server. If not declared, the settings are inherited from the parent scope. This allows you to customize or extend the JVM settings within every layer.

Let's take a look at the JVM declaration for a server group:

```
<server-groups>
       <server-group name="main-server-group" profile="default">
           <jvm name="default">
               <heap size="64m" max-size="512m"/>
           </jvm>
           <socket-binding-group ref="standard-sockets"/>
       </server-group>
       <server-group name="other-server-group" profile="default">
           <jvm name="default">
               <heap size="64m" max-size="512m"/>
           </jvm>
           <socket-binding-group ref="standard-sockets"/>
       </server-group>
</server-groups>
```

(See `domain/configuration/domain.xml`)

In this example the server group "main-server-group" declares a heap size of `64m` and a maximum heap size of `512m`. Any server that belongs to this group will inherit these settings. You can change these settings for the group as a whole, or a specific server or host:

```
<servers>
       <server name="server-one" group="main-server-group" auto-start="true">
           <jvm name="default"/>
       </server>
       <server name="server-two" group="main-server-group" auto-start="true">
           <jvm name="default">
               <heap size="64m" max-size="256m"/>
           </jvm>
           <socket-binding-group ref="standard-sockets" port-offset="150"/>
       </server>
       <server name="server-three" group="other-server-group" auto-start="false">
           <socket-binding-group ref="standard-sockets" port-offset="250"/>
       </server>
</servers>
```

(See domain/configuration/host.xml)

In this case, *server-two*, belongs to the *main-server-group* and inherits the JVM settings named *default*, but declares a lower maximum heap size.

```
[domain@localhost:9999 /] /host=local/server-config=server-two/jvm=default:read-resource
{
   "outcome" => "success",
   "result" => {
       "heap-size" => "64m",
       "max-heap-size" => "256m",
   }
}
```

Standalone Server

For a standalone sever you have to pass in the JVM settings either as command line arguments when executing the `$JBOSS_HOME/bin/standalone.sh` script, or by declaring them in `$JBOSS_HOME/bin/standalone.conf`. (For Windows users, the script to execute is `%JBOSS_HOME%/bin/standalone.bat` while the JVM settings can be declared in `%JBOSS_HOME%/bin/standalone.conf.bat`.)

5.9.5 Administrative audit logging

WildFly comes with audit logging built in for management operations affecting the management model. By default it is turned off. The information is output as JSON records.

The default configuration of audit logging in standalone.xml looks as follows:

```
<management>
        <security-realms>
...
        </security-realms>
        <audit-log>
            <formatters>
                <json-formatter name="json-formatter"/>
            </formatters>
            <handlers>
                <file-handler name="file" formatter="json-formatter" path="audit-log.log"
relative-to="jboss.server.data.dir"/>
            </handlers>
            <logger log-boot="true" log-read-only="true" enabled="false">
                <handlers>
                    <handler name="file"/>
                </handlers>
            </logger>
        </audit-log>
...
```

Looking at this via the CLI it looks like

```
[standalone@localhost:9990 /]
/core-service=management/access=audit:read-resource(recursive=true)
{
    "outcome" => "success",
    "result" => {
        "file-handler" => {"file" => {
            "formatter" => "json-formatter",
            "max-failure-count" => 10,
            "path" => "audit-log.log",
            "relative-to" => "jboss.server.data.dir"
        }},
        "json-formatter" => {"json-formatter" => {
            "compact" => false,
            "date-format" => "yyyy-MM-dd HH:mm:ss",
            "date-separator" => " - ",
            "escape-control-characters" => false,
            "escape-new-line" => false,
            "include-date" => true
        }},
        "logger" => {"audit-log" => {
            "enabled" => false,
            "log-boot" => true,
            "log-read-only" => false,
            "handler" => {"file" => {}}
        }},
        "syslog-handler" => undefined
    }
}
```

To enable it via CLI you need just

```
[standalone@localhost:9990 /]
/core-service=management/access=audit/logger=audit-log:write-attribute(name=enabled,value=true)
{"outcome" => "success"}
```

Audit data are stored in standalone/data/audit-log.log.

The audit logging subsystem has a lot of internal dependencies, and it logs operations changing, enabling and disabling its components. When configuring or changing things at runtime it is a good idea to make these changes as part of a CLI batch. For example if you are adding a syslog handler you need to add the handler and its information as one step. Similarly if you are using a file handler, and want to change its `path` and `relative-to` attributes, that needs to happen as one step.

JSON Formatter

The first thing that needs configuring is the formatter, we currently support outputting log records as JSON. You can define several formatters, for use with different handlers. A log record has the following format, and it is the formatter's job to format the data presented:

```
2013-08-12 11:01:12 - {
    "type" : "core",
    "r/o" : false,
    "booting" : false,
    "version" : "8.0.0.Alpha4",
    "user" : "$local",
    "domainUUID" : null,
    "access" : "NATIVE",
    "remote-address" : "127.0.0.1/127.0.0.1",
    "success" : true,
    "ops" : [JMX|WFLY8:JMX subsystem configuration],
        "operation" : "write-attribute",
        "name" : "enabled",
        "value" : true,
        "operation-headers" : {"caller-type" : "user"}
    }]
}
```

It includes an optional timestamp and then the following information in the json record

Field name	Description
`type`	This can have the values `core`, meaning it is a management operation, or `jmx` meaning it comes from the jmx subsystem (see the jmx subsystem for configuration of the jmx subsystem's audit logging)
`r/o`	`true` if the operation does not change the management model, `false` otherwise
`booting`	`true` if the operation was executed during the bootup process, `false` if it was executed once the server is up and running
`version`	The version number of the WildFly instance
`user`	The username of the authenticated user. In this case the operation has been logged via the CLI on the same machine as the running server, so the special `$local` user is used
`domainUUID`	An ID to link together all operations as they are propagated from the Doman Controller to it servers, slave Host Controllers, and slave Host Controller servers
`access`	This can have one of the following values: *`NATIVE` - The operation came in through the native management interface, for example the CLI *`HTTP` - The operation came in through the domain HTTP interface, for example the admin console *`JMX` - The operation came in through the JMX subsystem. See JMX for how to configure audit logging for JMX.
`remote-address`	The address of the client executing this operation
`success`	`true` if the operation succeeded, `false` if it was rolled back
`ops`	The operations being executed. This is a list of the operations serialized to JSON. At boot this will be all the operations resulting from parsing the xml. Once booted the list will typically just contain a single entry

The json formatter resource has the following attributes:

Attribute	Description
`include-date`	Boolan toggling whether or not to include the timestamp in the formatted log records
`date-separator`	A string containing characters to separate the date and the rest of the formatted log message. Will be ignored if `include-date=false`
`date-format`	The date format to use for the timestamp as understood by `java.text.SimpleDateFormat`. Will be ignored if `include-date=false`
`compact`	If true will format the JSON on one line. There may still be values containing new lines, so if having the whole record on one line is important, set escape-new-line or escape-control-characters to true
`escape-control-characters`	If `true` it will escape all control characters (ascii entries with a decimal value < 32) with the ascii code in octal, e.g. a new line becomes '#012'. If this is `true`, it will override `escape-new-line=false`
`escape-new-line`	If `true` it will escape all new lines with the ascii code in octal, e.g. "#012".

Handlers

A handler is responsible for taking the formatted data and logging it to a location. There are currently two types of handlers, File and Syslog. You can configure several of each type of handler and use them to log information.

File handler

The file handlers log the audit log records to a file on the server. The attributes for the file handler are

Attribute	Description	Read Only
`formatter`	The name of a JSON formatter to use to format the log records	`false`
`path`	The path of the audit log file	`false`
`relative-to`	The name of another previously named path, or of one of the standard paths provided by the system. If `relative-to` is provided, the value of the `path` attribute is treated as relative to the path specified by this attribute	`false`
`failure-count`	The number of logging failures since the handler was initialized	`true`
`max-failure-count`	The maximum number of logging failures before disabling this handler	`false`
`disabled-due-to-failure`	`true` if this handler was disabled due to logging failures	`true`

In our standard configuration `path=audit-log.log` and `relative-to=jboss.server.data.dir`, typically this will be `$JBOSS_HOME/standalone/data/audit-log.log`

Syslog handler

The default configuration does not have syslog audit logging set up. Syslog is a better choice for audit logging since you can log to a remote syslog server, and secure the authentication to happen over TLS with client certificate authentication. Syslog servers vary a lot in their capabilities so not all settings in this section apply to all syslog servers. We have tested with rsyslog.

The address for the syslog handler is `/core-service=management/access=audit/syslog-handler=*` and just like file handlers you can add as many syslog entries as you like. The syslog handler resources reference the main RFC's for syslog a fair bit, for reference they can be found at:
*http://www.ietf.org/rfc/rfc3164.txt
*http://www.ietf.org/rfc/rfc5424.txt
*http://www.ietf.org/rfc/rfc6587.txt

The syslog handler resource has the following attributes:

`formatter`	The name of a JSON formatter to use to format the log records	`false`
`failure-count`	The number of logging failures since the handler was initialized	`true`
`max-failure-count`	The maximum number of logging failures before disabling this handler	`false`
`disabled-due-to-failure`	`true` if this handler was disabled due to logging failures	`true`
`syslog-format`	Whether to set the syslog format to the one specified in RFC-5424 or RFC-3164	`false`
`max-length`	The maximum length in bytes a log message, including the header, is allowed to be. If undefined, it will default to 1024 bytes if the syslog-format is RFC3164, or 2048 bytes if the syslog-format is RFC5424.	`false`
`truncate`	Whether or not a message, including the header, should truncate the message if the length in bytes is greater than the maximum length. If set to false messages will be split and sent with the same header values	`false`

When adding a syslog handler you also need to add the protocol it will use to communicate with the syslog server. The valid choices for protocol are `UDP`,`TCP` and `TLS`. The protocol must be added at the same time as you add the syslog handler, or it will fail. Also, you can only add one protocol for the handler.

UDP

Configures the handler to use UDP to communicate with the syslog server. The address of the `UDP` resource is `/core-service=management/access=audit/syslog-handler=*/protocol=udp`. The attributes of the `UDP` resource are:

Attribute	Description
`host`	The host of the syslog server for the udp requests
`port`	The port of the syslog server listening for the udp requests

TCP

Configures the handler to use TCP to communicate with the syslog server. The address of the `TCP` resource is `/core-service=management/access=audit/syslog-handler=*/protocol=tcp`. The attributes of the `TCP` resource are:

Attribute	Description
`host`	The host of the syslog server for the tcp requests
`port`	The port of the syslog server listening for the tcp requests
`message-transfer`	The message transfer setting as described in section 3.4 of RFC-6587. This can either be OCTET_COUNTING as described in section 3.4.1 of RFC-6587, or NON_TRANSPARENT_FRAMING as described in section 3.4.1 of RFC-6587

TLS

Configures the handler to use TLC to communicate securely with the syslog server. The address of the `TLS` resource is `/core-service=management/access=audit/syslog-handler=*/protocol=tls`. The attributes of the `TLS` resource are the same as for `TCP`:

Attribute	Description
`host`	The host of the syslog server for the tls requests
`port`	The port of the syslog server listening for the tls requests
`message-transfer`	The message transfer setting as described in section 3.4 of RFC-6587. This can either be OCTET_COUNTING as described in section 3.4.1 of RFC-6587, or NON_TRANSPARENT_FRAMING as described in section 3.4.1 of RFC-6587

If the syslog server's TLS certificate is not signed by a certificate signing authority, you will need to set up a truststore to trust the certificate. The resource for the trust store is a child of the `TLS` resource, and the full address is `/core-service=management/access=audit/syslog-handler=*/protocol=tls/authenticati`. The attributes of the truststore resource are:

Attribute	Description
`keystore-password`	The password for the truststore
`keystore-path`	The path of the truststore
`keystore-relative-to`	The name of another previously named path, or of one of the standard paths provided by the system. If `keystore-relative-to` is provided, the value of the `keystore-path` attribute is treated as relative to the path specified by this attribute

TLS with Client certificate authentication.
If you have set up the syslog server to require client certificate authentication, when creating your handler you will also need to set up a client certificate store containing the certificate to be presented to the syslog server. The address of the client certificate store resource is `/core-service=management/access=audit/syslog-handler=*/protocol=tls/authenticati` and its attributes are:

Attribute	Description
`keystore-password`	The password for the keystore
`key-password`	The password for the keystore key
`keystore-path`	The path of the keystore
`keystore-relative-to`	The name of another previously named path, or of one of the standard paths provided by the system. If `keystore-relative-to` is provided, the value of the `keystore-path` attribute is treated as relative to the path specified by this attribute

Logger configuration

The final part that needs configuring is the logger for the management operations. This references one or more handlers and is configured at `/core-service=management/access=audit/logger=audit-log`. The attributes for this resource are:

Attribute	Description
`enabled`	`true` to enable logging of the management operations
`log-boot`	`true` to log the management operations when booting the server, `false` otherwise
`log-read-only`	If `true` all operations will be audit logged, if `false` only operations that change the model will be logged

Then which handlers are used to log the management operations are configured as `handler=*` children of the logger.

Domain Mode (host specific configuration)

In domain mode audit logging is configured for each host in its `host.xml` file. This means that when connecting to the DC, the configuration of the audit logging is under the host's entry, e.g. here is the default configuration:

```
[domain@localhost:9990 /]
/host=master/core-service=management/access=audit:read-resource(recursive=true)
{
    "outcome" => "success",
    "result" => {
        "file-handler" => {
            "host-file" => {
                "formatter" => "json-formatter",
                "max-failure-count" => 10,
                "path" => "audit-log.log",
                "relative-to" => "jboss.domain.data.dir"
            },
            "server-file" => {
                "formatter" => "json-formatter",
                "max-failure-count" => 10,
                "path" => "audit-log.log",
                "relative-to" => "jboss.server.data.dir"
            }
        },
        "json-formatter" => {"json-formatter" => {
            "compact" => false,
            "date-format" => "yyyy-MM-dd HH:mm:ss",
            "date-separator" => " - ",
            "escape-control-characters" => false,
            "escape-new-line" => false,
            "include-date" => true
        }},
        "logger" => {"audit-log" => {
            "enabled" => false,
            "log-boot" => true,
            "log-read-only" => false,
            "handler" => {"host-file" => {}}
        }},
        "server-logger" => {"audit-log" => {
            "enabled" => false,
            "log-boot" => true,
            "log-read-only" => false,
            "handler" => {"server-file" => {}}
        }},
        "syslog-handler" => undefined
    }
}
```

We now have two file handlers, one called `host-file` used to configure the file to log management operations on the host, and one called `server-file` used to log management operations executed on the servers. Then `logger=audit-log` is used to configure the logger for the host controller, referencing the `host-file` handler. `server-logger=audit-log` is used to configure the logger for the managed servers, referencing the `server-file` handler. The attributes for `server-logger=audit-log` are the same as for `server-logger=audit-log` in the previous section. Having the host controller and server loggers configured independently means we can control audit logging for managed servers and the host controller independently.

5.9.6 Canceling management operations

WildFly includes the ability to use the CLI to cancel management requests that are not proceeding normally.

The cancel-non-progressing-operation operation

The `cancel-non-progressing-operation` operation instructs the target process to find any operation that isn't proceeding normally and cancel it.

On a standalone server:

```
[standalone@localhost:9990 /]
/core-service=management/service=management-operations:cancel-non-progressing-operation
{
    "outcome" => "success",
    "result" => "-1155777943"
}
```

The result value is an internal identification number for the operation that was cancelled.

On a managed domain host controller, the equivalent resource is in the host=<hostname> portion of the management resource tree:

```
[domain@localhost:9990 /]
/host=host-a/core-service=management/service=management-operations:cancel-non-progressing-operatio
"outcome" => "success",
    "result" => "2156877946"
}
```

An operation can be cancelled on an individual managed domain server as well:

```
[domain@localhost:9990 /]
/host=host-a/server=server-one/core-service=management/service=management-operations:cancel-non-pr
"outcome" => "success",
    "result" => "6497786512"
}
```

An operation is considered to be not proceeding normally if it has been executing with the exclusive operation lock held for longer than 15 seconds. Read-only operations do not acquire the exclusive operation lock, so this operation will not cancel read-only operations. Operations blocking waiting for another operation to release the exclusive lock will also not be cancelled.

If there isn't any operation that is failing to proceed normally, there will be a failure response:

```
[standalone@localhost:9990 /]
/core-service=management/service=management-operations:cancel-non-progressing-operation
{
    "outcome" => "failed",
    "failure-description" => "WFLYDM0089: No operation was found that has been holding the
operation execution write lock for long than [15] seconds",
    "rolled-back" => true
}
```

The find-non-progressing-operation operation

To simply learn the id of an operation that isn't proceeding normally, but not cancel it, use the `find-non-progressing-operation` operation:

```
[standalone@localhost:9990 /]
/core-service=management/service=management-operations:find-non-progressing-operation
{
    "outcome" => "success",
    "result" => "-1155777943"
}
```

If there is no non-progressing operation, the outcome will still be `success` but the result will be `undefined`.

Once the id of the operation is known, the management resource for the operation can be examined to learn more about its status.

Examining the status of an active operation

There is a management resource for any currently executing operation that can be queried:

```
[standalone@localhost:9990 /]
/core-service=management/service=management-operations/active-operation=-1155777943:read-resource(
"outcome" => "success",
    "result" => {
        "access-mechanism" => "undefined",
        "address" => [
            ("deployment" => "example")
        ],
        "caller-thread" => "management-handler-thread - 24",
        "cancelled" => false,
        "exclusive-running-time" => 101918273645L,
        "execution-status" => "awaiting-stability",
        "operation" => "deploy",
        "running-time" => 101918279999L
    }
}
```

The response includes the following attributes:

Field	Meaning
access-mechanism	The mechanism used to submit a request to the server. NATIVE, JMX, HTTP
address	The address of the resource targeted by the operation. The value in the final element of the address will be '<hidden>' if the caller is not authorized to address the operation's target resource.
caller-thread	The name of the thread that is executing the operation.
cancelled	Whether the operation has been cancelled.
exclusive-running-time	Amount of time in nanoseconds the operation has been executing with the exclusive operation execution lock held, or -1 if the operation does not hold the exclusive execution lock.
execution-status	The current activity of the operation. See below for details.
operation	The name of the operation, or '<hidden>' if the caller is not authorized to address the operation's target resource.
running-time	Amount of time the operation has been executing, in nanoseconds.

The following are the values for the `execution-status` attribute:

Value	Meaning
executing	The caller thread is actively executing
awaiting-other-operation	The caller thread is blocking waiting for another operation to release the exclusive execution lock
awaiting-stability	The caller thread has made changes to the service container and is waiting for the service container to stabilize
completing	The operation is committed and is completing execution
rolling-back	The operation is rolling back

All currently executing operations can be viewed in one request using the `read-children-resources` operation:

```
[standalone@localhost:9990 /]
/core-service=management/service=management-operations:read-children-resources(child-type=active-o
"outcome" => "success",
    "result" => {"-1155777943" => {
        "access-mechanism" => "undefined",
        "address" => [
            ("deployment" => "example")
        ],
        "caller-thread" => "management-handler-thread - 24",
        "cancelled" => false,
        "exclusive-running-time" => 101918273645L,
        "execution-status" => "awaiting-stability",
        "operation" => "deploy",
        "running-time" => 101918279999L
    },
    {"-1246693202" => {
        "access-mechanism" => "undefined",
        "address" => [
            ("core-service" => "management"),
            ("service" => "management-operations")
        ],
        "caller-thread" => "management-handler-thread - 30",
        "cancelled" => false,
        "exclusive-running-time" => -1L,
        "execution-status" => "executing",
        "operation" => "read-children-resources",
        "running-time" => 3356000L
    }}
}
```

Canceling a specific operation

The `cancel-non-progressing-operation` operation is a convenience operation for identifying and canceling an operation. However, an administrator can examine the active-operation resources to identify any operation, and then directly cancel it by invoking the `cancel` operation on the resource for the desired operation.

```
[standalone@localhost:9990 /]
/core-service=management/service=management-operations/active-operation=-1155777943:cancel
{
    "outcome" => "success",
    "result" => undefined
}
```

Controlling operation blocking time

As an operation executes, the execution thread may block at various points, particularly while waiting for the service container to stabilize following any changes. Since an operation may be holding the exclusive execution lock while blocking, in WildFly execution behavior was changed to ensure that blocking will eventually time out, resulting in roll back of the operation.

The default blocking timeout is 300 seconds. This is intentionally long, as the idea is to only trigger a timeout when something has definitely gone wrong with the operation, without any false positives.

An administrator can control the blocking timeout for an individual operation by using the `blocking-timeout` operation header. For example, if a particular deployment is known to take an extremely long time to deploy, the default 300 second timeout could be increased:

```
[standalone@localhost:9990 /] deploy /tmp/mega.war --headers={blocking-timeout=450}
```

Note the blocking timeout is **not** a guaranteed maximum execution time for an operation. If it only a timeout that will be enforced at various points during operation execution.

5.9.7 Configuration file history

The management operations may modify the model. When this occurs the xml backing the model is written out again reflecting the latest changes. In addition a full history of the file is maintained. The history of the file goes in a separate directory under the configuration directory.

As mentioned in Command line parameters#parameters the default configuration file can be selected using a command-line parameter. For a standalone server instance the history of the active `standalone.xml` is kept in `jboss.server.config.dir`/standalone_xml_history (See Command line parameters#standalone_system_properties for more details). For a domain the active `domain.xml` and `host.xml` histories are kept in `jboss.domain.config.dir`/domain_xml_history and `jboss.domain.config.dir`/host_xml_history.

The rest of this section will only discuss the history for `standalone.xml`. The concepts are exactly the same for `domain.xml` and `host.xml`.

Within `standalone_xml_history` itself following a successful first time boot we end up with three new files:

- `standalone.initial.xml` - This contains the original configuration that was used the first time we successfully booted. This file will never be overwritten. You may of course delete the history directory and any files in it at any stage.
- `standalone.boot.xml` - This contains the original configuration that was used for the last successful boot of the server. This gets overwritten every time we boot the server successfully.
- `standalone.last.xml` - At this stage the contents will be identical to `standalone.boot.xml`. This file gets overwritten each time the server successfully writes the configuration, if there was an unexpected failure writing the configuration this file is the last known successful write.

`standalone_xml_history` contains a directory called `current` which should be empty. Now if we execute a management operation that modifies the model, for example adding a new system property using the CLI:

```
[standalone@localhost:9990 /] /system-property=test:add(value="test123")
{"outcome" => "success"}
```

What happens is:

- The original configuration file is backed up to `standalone_xml_history/current/standalone.v1.xml`. The next change to the model would result in a file called `standalone.v2.xml` etc. The 100 most recent of these files are kept.
- The change is applied to the original configuration file
- The changed original configuration file is copied to `standalone.last.xml`

When restarting the server, any existing `standalone_xml_history/current` directory is moved to a new timestamped folder within the `standalone_xml_history`, and a new `current` folder is created. These timestamped folders are kept for 30 days.

Snapshots

In addition to the backups taken by the server as described above you can manually take take snapshots which will be stored in the `snapshot` folder under the `_xml_history` folder, the automatic backups described above are subject to automatic house keeping so will eventually be automatically removed, the snapshots on the other hand can be entirely managed by the administrator.

You may also take your own snapshots using the CLI:

```
[standalone@localhost:9990 /] :take-snapshot
{
    "outcome" => "success",
    "result" => {"name" =>
"/Users/kabir/wildfly/standalone/configuration/standalone_xml_history/snapshot/20110630-172258657s
```

You can also use the CLI to list all the snapshots

```
[standalone@localhost:9990 /] :list-snapshots
{
    "outcome" => "success",
    "result" => {
        "directory" =>
"/Users/kabir/wildfly/standalone/configuration/standalone_xml_history/snapshot",
        "names" => [
            "20110630-165714239standalone.xml",
            "20110630-165821795standalone.xml",
            "20110630-170113581standalone.xml",
            "20110630-171411463standalone.xml",
            "20110630-171908397standalone.xml",
            "20110630-172258657standalone.xml"
        ]
    }
}
```

To delete a particular snapshot:

```
[standalone@localhost:9990 /] :delete-snapshot(name="20110630-165714239standalone.xml")
{"outcome" => "success"}
```

and to delete all snapshots:

```
[standalone@localhost:9990 /] :delete-snapshot(name="all")
{"outcome" => "success"}
```

In domain mode executing the snapshot operations against the root node will work against the domain model. To do this for a host model you need to navigate to the host in question:

```
[domain@localhost:9990 /] /host=master:list-snapshots
{
    "outcome" => "success",
    "result" => {
        "domain-results" => {"step-1" => {
            "directory" =>
"/Users/kabir/wildfly/domain/configuration/host_xml_history/snapshot",
            "names" => [
                "20110630-141129571host.xml",
                "20110630-172522225host.xml"
            ]
        }},
        "server-operations" => undefined
    }
}
```

Subsequent Starts

For subsequent server starts it may be desirable to take the state of the server back to one of the previously known states, for a number of items an abbreviated reverence to the file can be used:

Abreviation	Parameter	Description
`initial`	`--server-config=initial`	This will start the server using the initial configuration first used to start the server.
`boot`	`--server-config=boot`	This will use the configuration from the last successful boot of the server.
`last`	`--server-config=last`	This will start the server using the configuration backed up from the last successful save.
`v?`	`--server-config=v?`	This will server the _xml_history/current folder for the configuration where ? is the number of the backup to use.
`-?`	`--server-config=-?`	The server will be started after searching the snapshot folder for the configuration which matches this prefix.

In addition to this the `--server-config` parameter can always be used to specify a configuration relative to the `jboss.server.config.dir` and finally if no matching configuration is found an attempt to locate the configuration as an absolute path will be made.

5.10 Management API reference

This section is an in depth reference to the WildFly management API. Readers are encouraged to read the Management Clients and Core management concepts sections for fundamental background information, as well as the Management tasks and Domain Setup sections for key task oriented information. This section is meant as an in depth reference to delve into some of the key details.

5.10.1 Global operations

The WildFly management API includes a number of operations that apply to every resource.

The read-resource operation

Reads a management resource's attribute values along with either basic or complete information about any child resources. Supports the
following parameters, none of which are required:

- `recursive` – (boolean, default is `false`) – whether to include complete information about child resources, recursively.
- `recursive-depth` – (int) – The depth to which information about child resources should be included if `recursive` is `true`. If not set, the depth will be unlimited; i.e. all descendant resources will be included.
- `proxies` – (boolean, default is `false`) – whether to include remote resources in a recursive query (i.e. host level resources from slave Host Controllers in a query of the Domain Controller; running server resources in a query of a host).
- `include-runtime` – (boolean, default is `false`) – whether to include runtime attributes (i.e. those whose value does not come from the persistent configuration) in the response.
- `include-defaults` – (boolean, default is `true`) – whether to include in the result default values not set by users. Many attributes have a default value that will be used in the runtime if the users have not provided an explicit value. If this parameter is `false` the value for such attributes in the result will be `undefined`. If `true` the result will include the default value for such parameters.

The read-attribute operation

Reads the value of an individual attribute. Takes a single, required, parameter:

- `name` – (string) – the name of the attribute to read.
- `include-defaults` – (boolean, default is `true`) – whether to include in the result default values not set by users. Many attributes have a default value that will be used in the runtime if the users have not provided an explicit value. If this parameter is `false` the value for such attributes in the result will be `undefined`. If `true` the result will include the default value for such parameters.

The write-attribute operation

Writes the value of an individual attribute. Takes two required parameters:

- `name` – (string) – the name of the attribute to write.
- `value` – (type depends on the attribute being written) – the new value.

The undefine-attribute operation

Sets the value of an individual attribute to the `undefined` value, if such a value is allowed for the attribute. The operation will fail if the `undefined` value is not allowed. Takes a single required parameter:

- `name` – (string) – the name of the attribute to write.

The list-add operation

Adds an element to the value of a list attribute, adding the element to the end of the list unless the optional attribute `index` is passed:

- `name` – (string) – the name of the list attribute to add new value to.
- `value` – (type depends on the element being written) – the new element to be added to the attribute value.
- `index` – (int, optional) – index where in the list to add the new element. By default it is `undefined` meaning add at the end. Index is zero based.

This operation will fail if the specified attribute is not a list.

The list-remove operation

Removes an element from the value of a list attribute, either the element at a specified `index`, or the first element whose value matches a specified `value`:

- `name` – (string) – the name of the list attribute to add new value to.
- `value` – (type depends on the element being written, optional) – the element to be removed. Optional and ignored if `index` is specified.
- `index` – (int, optional) – index in the list whose element should be removed. By default it is `undefined`, meaning `value` should be specified.

This operation will fail if the specified attribute is not a list.

The list-get operation

Gets one element from a list attribute by its index

- `name` – (string) – the name of the list attribute
- `index` – (int, required) – index of element to get from list

This operation will fail if the specified attribute is not a list.

The list-clear operation

Empties the list attribute. It is different from `:undefine-attribute` as it results in attribute of type list with 0 elements, whereas `:undefine-attribute` results in an `undefined` value for the attribute

- `name` – (string) – the name of the list attribute

This operation will fail if the specified attribute is not a list.

The map-put operation

Adds an key/value pair entry to the value of a map attribute:

- `name` – (string) – the name of the map attribute to add the new entry to.
- `key` – (string) – the key of the new entry to be added.
- `value` – (type depends on the entry being written) – the value of the new entry to be added to the attribute value.

This operation will fail if the specified attribute is not a map.

The map-remove operation

Removes an entry from the value of a map attribute:

- `name` – (string) – the name of the map attribute to remove the new entry from.
- `key` – (string) – the key of the entry to be removed.

This operation will fail if the specified attribute is not a map.

The map-get operation

Gets the value of one entry from a map attribute

- `name` – (string) – the name of the map attribute
- `key` – (string) – the key of the entry.

This operation will fail if the specified attribute is not a map.

The map-clear operation

Empties the map attribute. It is different from `:undefine-attribute` as it results in attribute of type map with 0 entries, whereas `:undefine-attribute` results in an `undefined` value for the attribute

- `name` – (string) – the name of the map attribute

This operation will fail if the specified attribute is not a map.

The read-resource-description operation

Returns the description of a resource's attributes, types of children and, optionally, operations. Supports the following parameters, none of which are required:

- `recursive` – (boolean, default is `false`) – whether to include information about child resources, recursively.
- `proxies` – (boolean, default is `false`) – whether to include remote resources in a recursive query (i.e. host level resources from slave Host Controllers in a query of the Domain Controller; running server resources in a query of a host)
- `operations` – (boolean, default is `false`) – whether to include descriptions of the resource's operations
- `inherited` – (boolean, default is `true`) – if `operations` is `true`, whether to include descriptions of operations inherited from higher level resources. The global operations described in this section are themselves inherited from the root resource, so the primary effect of setting `inherited` to `false` is to exclude the descriptions of the global operations from the output.

See Description of the Management Model for details on the result of this operation.

The read-operation-names operation

Returns a list of the names of all the operations the resource supports. Takes no parameters.

The read-operation-description operation

Returns the description of an operation, along with details of its parameter types and its return value. Takes a single, required, parameter:

- `name` – (string) – the name of the operation

See Description of the Management Model for details on the result of this operation.

The read-children-types operation

Returns a list of the types of child resources the resource supports. Takes two optional parameters:

- `include-aliases` – (boolean, default is `false`) – whether to include alias children (i.e. those which are aliases of other sub-resources) in the response.
- `include-singletons` – (boolean, default is `false`) – whether to include singleton children (i.e. those are children that acts as resource aggregate and are registered with a wildcard name) in the response wildfly-dev discussion around this topic.

The read-children-names operation

Returns a list of the names of all child resources of a given type. Takes a single, required, parameter:

- `child-type` – (string) – the name of the type

The read-children-resources operation

Returns information about all of a resource's children that are of a given type. For each child resource, the returned information is equivalent to executing the `read-resource` operation on that resource. Takes the following parameters, of which only {{child-type} is required:

- `child-type` – (string) – the name of the type of child resource
- `recursive` – (boolean, default is `false`) – whether to include complete information about child resources, recursively.
- `recursive-depth` – (int) – The depth to which information about child resources should be included if `recursive` is {{true}. If not set, the depth will be unlimited; i.e. all descendant resources will be included.
- `proxies` – (boolean, default is `false`) – whether to include remote resources in a recursive query (i.e. host level resources from slave Host Controllers in a query of the Domain Controller; running server resources in a query of a host)
- `include-runtime` – (boolean, default is `false`) – whether to include runtime attributes (i.e. those whose value does not come from the persistent configuration) in the response.
- `include-defaults` – (boolean, default is `true`) – whether to include in the result default values not set by users. Many attributes have a default value that will be used in the runtime if the users have not provided an explicit value. If this parameter is `false` the value for such attributes in the result will be `undefined`. If `true` the result will include the default value for such parameters.

The read-attribute-group operation

Returns a list of attributes of a type for a given attribute group name. For each attribute the returned information is equivalent to executing the `read-attribute` operation of that resource. Takes the following parameters, of which only {{name} is required:

- `name` – (string) – the name of the attribute group to read.
- `include-defaults` – (boolean, default is `true`) – whether to include in the result default values not set by users. Many attributes have a default value that will be used in the runtime if the users have not provided an explicit value. If this parameter is `false` the value for such attributes in the result will be `undefined`. If `true` the result will include the default value for such parameters.
- `include-runtime` – (boolean, default is `false`) – whether to include runtime attributes (i.e. those whose value does not come from the persistent configuration) in the response.
- `include-aliases` – (boolean, default is `false`) – whether to include alias attributes (i.e. those which are alias of other attributes) in the response.

The read-attribute-group-names operation

Returns a list of attribute groups names for a given type. Takes no parameters.

Standard Operations

Besides the global operations described above, by convention nearly every resource should expose an `add` operation and a `remove` operation. Exceptions to this convention are the root resource, and resources that do not store persistent configuration and are created dynamically at runtime (e.g. resources representing the JVM's platform mbeans or resources representing aspects of the running state of a deployment.)

The add operation

The operation that creates a new resource must be named `add`. The operation may take zero or more parameters; what those parameters are depends on the resource being created.

The remove operation

The operation that removes an existing resource must be named `remove`. The operation should take no parameters.

5.10.2 Detyped management and the jboss-dmr library

The management model exposed by WildFly is very large and complex. There are dozens, probably hundreds of logical concepts involved – hosts, server groups, servers, subsystems, datasources, web connectors, and on and on – each of which in a classic objected oriented API design could be represented by a Java *type* (i.e. a Java class or interface.) However, a primary goal in the development of WildFly's native management API was to ensure that clients built to use the API had as few compile-time and run-time dependencies on JBoss-provided classes as possible, and that the API exposed by those libraries be powerful but also simple and stable. A management client running with the management libraries created for an earlier version of WildFly should still work if used to manage a later version domain. The management client libraries needed to be *forward compatible*.

It is highly unlikely that an API that consists of hundreds of Java types could be kept forward compatible. Instead, the WildFly management API is a *detyped* API. A detyped API is like decaffeinated coffee – it still has a little bit of caffeine, but not enough to keep you awake at night. WildFly's management API still has a few Java types in it (it's impossible for a Java library to have no types!) but not enough to keep you (or us) up at night worrying that your management clients won't be forward compatible.

A detyped API works by making it possible to build up arbitrarily complex data structures using a small number of Java types. All of the parameter values and return values in the API are expressed using those few types. Ideally, most of the types are basic JDK types, like `java.lang.String`, `java.lang.Integer`, etc. In addition to the basic JDK types, WildFly's detyped management API uses a small library called **jboss-dmr**. The purpose of this section is to provide a basic overview of the jboss-dmr library.

Even if you don't use jboss-dmr directly (probably the case for all but a few users), some of the information in this section may be useful. When you invoke operations using the application server's Command Line Interface, the return values are just the text representation of of a jboss-dmr `ModelNode`. If your CLI commands require complex parameter values, you may yourself end up writing the text representation of a `ModelNode`. And if you use the HTTP management API, all response bodies as well as the request body for any POST will be a JSON representation of a `ModelNode`.

The source code for jboss-dmr is available on Github. The maven coordinates for a jboss-dmr release are `org.jboss.jboss-dmr:jboss-dmr`.

ModelNode and ModelType

The public API exposed by jboss-dmr is very simple: just three classes, one of which is an enum!

The primary class is `org.jboss.dmr.ModelNode`. A `ModelNode` is essentially just a wrapper around some *value*; the value is typically some basic JDK type. A `ModelNode` exposes a `getType()` method. This method returns a value of type `org.jboss.dmr.ModelType`, which is an enum of all the valid types of values. And that's 95% of the public API; a class and an enum. (We'll get to the third class, `Property`, below.)

Basic ModelNode manipulation

To illustrate how to work with `ModelNode` s, we'll use the Beanshell scripting library. We won't get into many details of beanshell here; it's a simple and intuitive tool and hopefully the following examples are as well.

We'll start by launching a beanshell interpreter, with the jboss-dmr library available on the classpath. Then we'll tell beanshell to import all the jboss-dmr classes so they are available for use:

```
$ java -cp bsh-2.0b4.jar:jboss-dmr-1.0.0.Final.jar bsh.Interpreter
BeanShell 2.0b4 - by Pat Niemeyer (pat@pat.net)
bsh % import org.jboss.dmr.*;
bsh %
```

Next, create a `ModelNode` and use the beanshell `print` function to output what type it is:

```
bsh % ModelNode node = new ModelNode();
bsh % print(node.getType());
UNDEFINED
```

A new `ModelNode` has no value stored, so its type is `ModelType.UNDEFINED`.

Use one of the overloaded `set` method variants to assign a node's value:

```
bsh % node.set(1);
bsh % print(node.getType());
INT
bsh % node.set(true);
bsh % print(node.getType());
BOOLEAN
bsh % node.set("Hello, world");
bsh % print(node.getType());
STRING
```

Use one of the `asXXX()` methods to retrieve the value:

```
bsh % node.set(2);
bsh % print(node.asInt());
2
bsh % node.set("A string");
bsh % print(node.asString());
A string
```

`ModelNode` will attempt to perform type conversions when you invoke the `asXXX` methods:

```
bsh % node.set(1);
bsh % print(node.asString());
1
bsh % print(node.asBoolean());
true
bsh % node.set(0);
bsh % print(node.asBoolean());
false
bsh % node.set("true");
bsh % print(node.asBoolean());
true
```

Not all type conversions are possible:

```
bsh % node.set("A string");
bsh % print(node.asInt());
// Error: // Uncaught Exception: Method Invocation node.asInt : at Line: 20 : in file: <unknown
file> : node .asInt ( )

Target exception: java.lang.NumberFormatException: For input string: "A string"

java.lang.NumberFormatException: For input string: "A string"
 at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
 at java.lang.Integer.parseInt(Integer.java:449)
 at java.lang.Integer.parseInt(Integer.java:499)
 at org.jboss.dmr.StringModelValue.asInt(StringModelValue.java:61)
 at org.jboss.dmr.ModelNode.asInt(ModelNode.java:117)
        ....
```

The `ModelNode.getType()` method can be used to ensure a node has an expected value type before attempting a type conversion.

One `set` variant takes another `ModelNode` as its argument. The value of the passed in node is copied, so there is no shared state between the two model nodes:

```
bsh % node.set("A string");
bsh % ModelNode another = new ModelNode();
bsh % another.set(node);
bsh % print(another.asString());
A string
bsh % node.set("changed");
bsh % print(node.asString());
changed
bsh % print(another.asString());
A string
```

A `ModelNode` can be cloned. Again, there is no shared state between the original node and its clone:

```
bsh % ModelNode clone = another.clone();
bsh % print(clone.asString());
A string
bsh % another.set(42);
bsh % print(another.asString());
42
bsh % print(clone.asString());
A string
```

Use the `protect()` method to make a `ModelNode` immutable:

```
bsh % clone.protect();
bsh % clone.set("A different string");
// Error: // Uncaught Exception: Method Invocation clone.set : at Line: 15 : in file: <unknown
file> : clone .set ( "A different string" )

Target exception: java.lang.UnsupportedOperationException

java.lang.UnsupportedOperationException
 at org.jboss.dmr.ModelNode.checkProtect(ModelNode.java:1441)
 at org.jboss.dmr.ModelNode.set(ModelNode.java:351)
        ....
```

Lists

The above examples aren't particularly interesting; if all we can do with a `ModelNode` is wrap a simple Java primitive, what use is that? However, a `ModelNode`'s value can be more complex than a simple primitive, and using these more complex types we can build complex data structures. The first more complex type is `ModelType.LIST`.

Use the `add` methods to initialize a node's value as a list and add to the list:

```
bsh % ModelNode list = new ModelNode();
bsh % list.add(5);
bsh % list.add(10);
bsh % print(list.getType());
LIST
```

Use `asInt()` to find the size of the list:

```
bsh % print(list.asInt());
2
```

Use the overloaded `get` method variant that takes an int param to retrieve an item. The item is returned as a `ModelNode`:

```
bsh % ModelNode child = list.get(1);
bsh % print(child.asInt());
10
```

Elements in a list need not all be of the same type:

```
bsh % list.add("A string");
bsh % print(list.get(1).getType());
INT
bsh % print(list.get(2).getType());
STRING
```

Here's one of the trickiest things about jboss-dmr: *The* `get` *methods actually mutate state; they are not "read-only"*. For example, calling `get` with an index that does not exist yet in the list will actually create a child of type `ModelType.UNDEFINED` at that index (and will create UNDEFINED children for any intervening indices.)

```
bsh % ModelNode four = list.get(4);
bsh % print(four.getType());
UNDEFINED
bsh % print(list.asInt());
6
```

Since the `get` call always returns a `ModelNode` and never `null` it is safe to manipulate the return value:

```
bsh % list.get(5).set(30);
bsh % print(list.get(5).asInt());
30
```

That's not so interesting in the above example, but later on with node of type `ModelType.OBJECT` we'll see how that kind of method chaining can let you build up fairly complex data structures with a minimum of code.

Use the `asList()` method to get a `List<ModelNode>` of the children:

```
bsh % for (ModelNode element : list.asList()) {
print(element.getType());
}
INT
INT
STRING
UNDEFINED
UNDEFINED
INT
```

The `asString()` and `toString()` methods provide slightly differently formatted text representations of a `ModelType.LIST` node:

```
bsh % print(list.asString());
[5,10,"A string",undefined,undefined,30]
bsh % print(list.toString());
[
    5,
    10,
    "A string",
    undefined,
    undefined,
    30
]
```

Finally, if you've previously used `set` to assign a node's value to some non-list type, you cannot use the `add` method:

```
bsh % node.add(5);
// Error: // Uncaught Exception: Method Invocation node.add : at Line: 18 : in file: <unknown
file> : node .add ( 5 )

Target exception: java.lang.IllegalArgumentException

java.lang.IllegalArgumentException
 at org.jboss.dmr.ModelValue.addChild(ModelValue.java:120)
 at org.jboss.dmr.ModelNode.add(ModelNode.java:1007)
 at org.jboss.dmr.ModelNode.add(ModelNode.java:761)
        ...
```

You can, however, use the `setEmptyList()` method to change the node's type, and then use `add`:

```
bsh % node.setEmptyList();
bsh % node.add(5);
bsh % print(node.toString());
[5]
```

Properties

The third public class in the jboss-dmr library is `org.jboss.dmr.Property`. A `Property` is a `String => ModelNode` tuple.

```
bsh % Property prop = new Property("stuff", list);
bsh % print(prop.toString());
org.jboss.dmr.Property@79a5f739
bsh % print(prop.getName());
stuff
bsh % print(prop.getValue());
[
    5,
    10,
    "A string",
    undefined,
    undefined,
    30
]
```

The property can be passed to `ModelNode.set`:

```
bsh % node.set(prop);
bsh % print(node.getType());
PROPERTY
```

The text format for a node of `ModelType.PROPERTY` is:

```
bsh % print(node.toString());
("stuff" => [
    5,
    10,
    "A string",
    undefined,
    undefined,
    30
])
```

Directly instantiating a `Property` via its constructor is not common. More typically one of the two argument `ModelNode.add` or `ModelNode.set` variants is used. The first argument is the property name:

```
bsh % ModelNode simpleProp = new ModelNode();
bsh % simpleProp.set("enabled", true);
bsh % print(simpleProp.toString());
("enabled" => true)
bsh % print(simpleProp.getType());
PROPERTY
bsh % ModelNode propList = new ModelNode();
bsh % propList.add("min", 1);
bsh % propList.add("max", 10);
bsh % print(propList.toString());
[
    ("min" => 1),
    ("max" => 10)
]
bsh % print(propList.getType());
LIST
bsh % print(propList.get(0).getType());
PROPERTY
```

The `asPropertyList()` method provides easy access to a `List<Property>`:

```
bsh % for (Property prop : propList.asPropertyList()) {
print(prop.getName() + " = " + prop.getValue());
}
min = 1
max = 10
```

ModelType.OBJECT

The most powerful and most commonly used complex value type in jboss-dmr is `ModelType.OBJECT`. A `ModelNode` whose value is `ModelType.OBJECT` internally maintains a `Map<String, ModelNode`.

Use the `get` method variant that takes a string argument to add an entry to the map. If no entry exists under the given name, a new entry is added with a the value being a `ModelType.UNDEFINED` node. The node is returned:

```
bsh % ModelNode range = new ModelNode();
bsh % ModelNode min = range.get("min");
bsh % print(range.toString());
{"min" => undefined}
bsh % min.set(2);
bsh % print(range.toString());
{"min" => 2}
```

Again **it is important to remember that the `get` operation may mutate the state of a model node by adding a new entry.** *It is not a read-only operation.*

Since `get` will never return `null`, a common pattern is to use method chaining to create the key/value pair:

```
bsh % range.get("max").set(10);
bsh % print(range.toString());
{
    "min" => 2,
    "max" => 10
}
```

A call to `get` passing an already existing key will of course return the same model node as was returned the first time `get` was called with that key:

```
bsh % print(min == range.get("min"));
true
```

Multiple parameters can be passed to `get`. This is a simple way to traverse a tree made up of `ModelType.OBJECT` nodes. Again, `get` may mutate the node on which it is invoked; e.g. it will actually create the tree if nodes do not exist. This next example uses a workaround to get beanshell to handle the overloaded `get` method that takes a variable number of arguments:

```
bsh % String[] varargs = { "US", "Missouri", "St. Louis" };
bsh % salesTerritories.get(varargs).set("Brian");
bsh % print(salesTerritories.toString());
{"US" => {"Missouri" => {"St. Louis" => "Brian"}}}
```

The normal syntax would be:

```
salesTerritories.get("US", "Missouri", "St. Louis").set("Brian");
```

The key/value pairs in the map can be accessed as a `List<Property`:

```
bsh % for (Property prop : range.asPropertyList()) {
print(prop.getName() + " = " + prop.getValue());
}
min = 2
```

The semantics of the backing map in a node of `ModelType.OBJECT` are those of a `LinkedHashMap`. The map remembers the order in which key/value pairs are added. This is relevant when iterating over the pairs after calling `asPropertyList()` and for controlling the order in which key/value pairs appear in the output from `toString()`.

Since the `get` method will actually mutate the state of a node if the given key does not exist, `ModelNode` provides a couple methods to let you check whether the entry is there. The `has` method simply does that:

```
bsh % print(range.has("unit"));
false
bsh % print(range.has("min"));
true
```

Very often, the need is to not only know whether the key/value pair exists, but whether the value is defined (i.e. not `ModelType.UNDEFINED`. This kind of check is analogous to checking whether a field in a Java class has a null value. The `hasDefined` lets you do this:

```
bsh % print(range.hasDefined("unit"));
false
bsh % // Establish an undefined child 'unit';
bsh % range.get("unit");
bsh % print(range.toString());
{
    "min" => 2,
    "max" => 10,
    "unit" => undefined
}
bsh % print(range.hasDefined("unit"));
false
bsh % range.get("unit").set("meters");
bsh % print(range.hasDefined("unit"));
true
```

ModelType.EXPRESSION

A value of type `ModelType.EXPRESSION` is stored as a string, but can later be *resolved* to different value. The string has a special syntax that should be familiar to those who have used the system property substitution feature in previous JBoss AS releases.

```
[<prefix>][${<system-property-name>[:<default-value>]}][<suffix>]*
```

For example:

```
${queue.length}
http://${host}
http://${host:localhost}:${port:8080}/index.html
```

Use the `setExpression` method to set a node's value to type expression:

```
bsh % ModelNode expression = new ModelNode();
bsh % expression.setExpression("${queue.length}");
bsh % print(expression.getType());
EXPRESSION
```

Calling `asString()` returns the same string that was input:

```
bsh % print(expression.asString());
${queue.length}
```

However, calling `toString()` tells you that this node's value is not of `ModelType.STRING`:

```
bsh % print(expression.toString());
expression "${queue.length}"
```

When the `resolve` operation is called, the string is parsed and any embedded system properties are resolved against the JVM's current system property values. A new `ModelNode` is returned whose value is the resolved string:

```
bsh % System.setProperty("queue.length", "10");
bsh % ModelNode resolved = expression.resolve();
bsh % print(resolved.asInt());
10
```

Note that the type of the `ModelNode` returned by `resolve()` is `ModelType.STRING`:

```
bsh % print(resolved.getType());
STRING
```

The `resolved.asInt()` call in the previous example only worked because the string "10" happens to be convertible into the int 10.

Calling `resolve()` has no effect on the value of the node on which the method is invoked:

```
bsh % resolved = expression.resolve();
bsh % print(resolved.toString());
"10"
bsh % print(expression.toString());
expression "${queue.length}"
```

If an expression cannot be resolved, `resolve` just uses the original string. The string can include more than one system property substitution:

```
bsh % expression.setExpression("http://${host}:${port}/index.html");
bsh % resolved = expression.resolve();
bsh % print(resolved.asString());
http://${host}:${port}/index.html
```

The expression can optionally include a default value, separated from the name of the system property by a colon:

```
bsh % expression.setExpression("http://${host:localhost}:${port:8080}/index.html");
bsh % resolved = expression.resolve();
bsh % print(resolved.asString());
http://localhost:8080/index.html
```

Actually including a system property substitution in the expression is not required:

```
bsh % expression.setExpression("no system property");
bsh % resolved = expression.resolve();
bsh % print(resolved.asString());
no system property
bsh % print(expression.toString());
expression "no system property"
```

The `resolve` method works on nodes of other types as well; it returns a copy without attempting any real resolution:

```
bsh % ModelNode basic = new ModelNode();
bsh % basic.set(10);
bsh % resolved = basic.resolve();
bsh % print(resolved.getType());
INT
bsh % resolved.set(5);
bsh % print(resolved.asInt());
5
bsh % print(basic.asInt());
10
```

ModelType.TYPE

You can also pass one of the values of the `ModelType` enum to `set`:

```
bsh % ModelNode type = new ModelNode();
bsh % type.set(ModelType.LIST);
bsh % print(type.getType());
TYPE
bsh % print(type.toString());
LIST
```

This is useful when using a `ModelNode` data structure to describe another `ModelNode` data structure.

Full list of ModelNode types

BIG_DECIMAL
BIG_INTEGER
BOOLEAN
BYTES
DOUBLE
EXPRESSION
INT
LIST
LONG
OBJECT
PROPERTY
STRING
TYPE
UNDEFINED

Text representation of a ModelNode

TODO – document the grammar

JSON representation of a ModelNode

TODO – document the grammar

5.10.3 Description of the Management Model

A detailed description of the resources, attributes and operations that make up the management model provided by an individual WildFly instance or by any Domain Controller or slave Host Controller process can be queried using the `read-resource-description`, `read-operation-names`, `read-operation-description` and `read-child-types` operations described in the Global operations section. In this section we provide details on what's included in those descriptions.

Description of the WildFly Managed Resources

All portions of the management model exposed by WildFly are addressable via an ordered list of key/value pairs. For each addressable *Management Resource*, the following descriptive information will be available:

- `description` – String – text description of this portion of the model
- `min-occurs` – int, either 0 or 1 – Minimum number of resources of this type that must exist in a valid model. If not present, the default value is 0.
- `max-occurs` – int – Maximum number of resources of this type that may exist in a valid model. If not present, the default value depends upon the value of the final key/value pair in the address of the described resource. If this value is '*', the default value is Integer.MAX_VALUE, i.e. there is no limit. If this value is some other string, the default value is 1.
- `attributes` – Map of String (the attribute name) to complex structure – the configuration attributes available in this portion of the model. See below for the representation of each attribute.
- `operations` – Map of String (the operation name) to complex structure – the operations that can be targetted at this address. See below for the representation of each operation.
- `children` – Map of String (the type of child) to complex structure – the relationship of this portion of the model to other addressable portions of the model. See below for the representation of each child relationship.
- `head-comment-allowed` – boolean – indicates whether this portion of the model can store an XML comment that would be written in the persistent form of the model (e.g. domain.xml) before the start of the XML element that represents this portion of the model. This item is optional, and if not present defaults to true. (Note: storing XML comments in the in-memory model is not currently supported. This description key is for future use.)
- `tail-comment-allowed` – boolean – similar to `head-comment-allowed`, but indicates whether a comment just before the close of the XML element is supported. A tail comment can only be supported if the element has child elements, in which case a comment can be inserted between the final child element and the element's closing tag. This item is optional, and if not present defaults to true. (Note: storing XML comments in the in-memory model is not currently supported. This description key is for future use.)

For example:

```
{
     "description => "A manageable resource",
     "tail-comment-allowed" => false,
     "attributes" => {
          "foo" => {
               .... details of attribute foo
          }
      },
     "operations" => {
          "start" => {
               .... details of the start operation
          }
      },
     "children" => {
          "bar" => {
               .... details of the relationship with children of type "bar"
          }
      }
}
```

Description of an Attribute

An attribute is a portion of the management model that is not directly addressable. Instead, it is conceptually a property of an addressable management resource. For each attribute in the model, the following descriptive information will be available:

- `description` – String – text description of the attribute
- `type` – `org.jboss.dmr.ModelType` – the type of the attribute value. One of the enum values BIG_DECIMAL, BIG_INTEGER, BOOLEAN, BYTES, DOUBLE, INT, LIST, LONG, OBJECT, PROPERTY, STRING. Most of these are self-explanatory. An OBJECT will be represented in the detyped model as a map of string keys to values of some other legal type, conceptually similar to a `javax.management.openmbean.CompositeData`. A PROPERTY is a single key/value pair, where the key is a string, and the value is of some other legal type.
- `value-type` – ModelType or complex structure – Only present if type is LIST or OBJECT. If all elements in the LIST or all the values of the OBJECT type are of the same type, this will be one of the ModelType enums BIG_DECIMAL, BIG_INTEGER, BOOLEAN, BYTES, DOUBLE, INT, LONG, STRING. Otherwise, `value-type` will detail the structure of the attribute value, enumerating the value's fields and the type of their value. So, an attribute with a `type` of LIST and a `value-type` value of `ModelType.STRING` is analogous to a Java `List<String>`, while one with a `value-type` value of `ModelType.INT` is analogous to a Java `List<Integer>`. An attribute with a `type` of OBJECT and a `value-type` value of `ModelType.STRING` is analogous to a Java `Map<String, String>`. An attribute with a `type` of OBJECT and a `value-type` whose value is not of type `ModelType` represents a fully-defined complex object, with the object's legal fields and their values described.
- `expressions-allowed` – boolean – indicates whether the value of the attribute may be of type `ModelType.EXPRESSION`, instead of its standard type (see `type` and `value-type` above for discussion of an attribute's standard type.) A value of ModelType.EXPRESSION contains a system-property substitution expression that the server will resolve against the server-side system property map before using the value. For example, an attribute named max-threads may have an expression value of `${example.pool.max-threads:10}` instead of just 10. Default value if not present is false.
- `required` – boolean – true if the attribute must have a defined value in a representation of its portion of the model unless another attribute included in a list of `alternatives` is defined; false if it may be undefined (implying a null value) even in the absence of alternatives. If not present, true is the default.
- `nillable` – boolean – true if the attribute might not have a defined value in a representation of its portion of the model. A nillable attribute may
  be undefined either because it is not `required` or because it is required but has `alternatives` and one of the alternatives is defined.
- `storage` – String – Either "configuration" or "runtime". If "configuration", the attribute's value is stored as part of the persistent configuration (e.g. in domain.xml, host.xml or standalone.xml.) If "runtime" the attribute's value is not stored in the persistent configuration; the value only exists as long as the resource is running.

- `access-type` – String – One of "read-only", "read-write" or "metric". Whether an attribute value can be written, or can only read. A "metric" is a read-only attribute whose value is not stored in the persistent configuration, and whose value may change due to activity on the server. If an attribute is "read-write", the resource will expose an operation named "write-attribute" whose "name" parameter will accept this attribute's name and whose "value" parameter will accept a valid value for this attribute. That operation will be the standard means of updating this attribute's value.
- `restart-required` – String – One of "no-services", "all-services", "resource-services" or "jvm". Only relevant to attributes whose access-type is read-write. Indicates whether execution of a write-attribute operation whose name parameter specifies this attribute requires a restart of services (or an entire JVM) in order for the change to take effect in the runtime . See discussion of "Applying Updates to Runtime Services" below. Default value is "no-services".
- `default` – the default value for the attribute that will be used in runtime services if the attribute is not explicitly defined and no other attributes listed as `alternatives` are defined.
- `alternatives` – List of string – Indicates an exclusive relationship between attributes. If this attribute is defined, the other attributes listed in this descriptor's value should be undefined, even if their `required` descriptor says true; i.e. the presence of this attribute satisfies the requirement. Note that an attribute that is not explicitly configured but has a `default` value is still regarded as not being defined for purposes of checking whether the exclusive relationship has been violated. Default is undefined; i.e. this does not apply to most attributes.
- `requires` – List of string – Indicates that if this attribute has a value (other than undefined), the other attributes listed in this descriptor's value must also have a value, even if their required descriptor says false. This would typically be used in conjunction with alternatives. For example, attributes "a" and "b" are required, but are alternatives to each other; "c" and "d" are optional. But "b" requires "c" and "d", so if "b" is used, "c" and "d" must also be defined. Default is undefined; i.e. this does not apply to most attributes.
- `capability-reference` – string – if defined indicates that this attribute's value specifies the dynamic portion of the name of the specified capability provided by another resource. This indicates the attribute is a reference to another area of the management model. (Note that at present some attributes that reference other areas of the model may not provide this information.)
- `head-comment-allowed` – boolean – indicates whether the model can store an XML comment that would be written in the persistent form of the model (e.g. domain.xml) before the start of the XML element that represents this attribute. This item is optional, and if not present defaults to false. (This is a different default from what is used for an entire management resource, since model attributes often map to XML attributes, which don't allow comments.) (Note: storing XML comments in the in-memory model is not currently supported. This description key is for future use.)
- `tail-comment-allowed` – boolean – similar to head-comment-allowed, but indicates whether a comment just before the close of the XML element is supported. A tail comment can only be supported if the element has child elements, in which case a comment can be inserted between the final child element and the element's closing tag. This item is optional, and if not present defaults to false. (This is a different default from what is used for an entire management resource, since model attributes often map to XML attributes, which don't allow comments.) (Note: storing XML comments in the in-memory model is not currently supported. This description key is for future use.)
- arbitrary key/value pairs that further describe the attribute value, e.g. "max" => 2. See "Arbitrary Descriptors" below.

Some examples:

```
"foo" => {
     "description" => "The foo",
     "type" => INT,
     "max" => 2
}
```

```
"bar" => {
     "description" => "The bar",
     "type" => OBJECT,
     "value-type" => {
          "size" => INT,
          "color" => STRING
     }
}
```

Description of an Operation

A management resource may have operations associated with it. The description of an operation will include the following information:

- `operation-name` – String – the name of the operation
- `description` – String – text description of the operation
- `request-properties` – Map of String to complex structure – description of the parameters of the operation. Keys are the names of the parameters, values are descriptions of the parameter value types. See below for details on the description of parameter value types.
- `reply-properties` – complex structure, or empty – description of the return value of the operation, with an empty node meaning void. See below for details on the description of operation return value types.
- `restart-required` – String – One of "no-services", "all-services", "resource-services" or "jvm". Indicates whether the operation makes a configuration change that requires a restart of services (or an entire JVM) in order for the change to take effect in the runtime. See discussion of "Applying Updates to Runtime Services" below. Default value is "no-services".

Description of an Operation Parameter or Return Value

- `description` – String – text description of the parameter or return value
- `type` – `org.jboss.dmr.ModelType` – the type of the parameter or return value. One of the enum values BIG_DECIMAL, BIG_INTEGER, BOOLEAN, BYTES, DOUBLE, INT, LIST, LONG, OBJECT, PROPERTY, STRING.

- `value-type` – `ModelType` or complex structure – Only present if type is LIST or OBJECT. If all elements in the LIST or all the values of the OBJECT type are of the same type, this will be one of the ModelType enums BIG_DECIMAL, BIG_INTEGER, BOOLEAN, BYTES, DOUBLE, INT, LIST, LONG, PROPERTY, STRING. Otherwise, value-type will detail the structure of the attribute value, enumerating the value's fields and the type of their value.So, a parameter with a `type` of LIST and a `value-type` value of `ModelType.STRING` is analogous to a Java `List<String>`, while one with a `value-type` value of `ModelType.INT` is analogous to a Java `List<Integer>`. A parameter with a `type` of OBJECT and a `value-type` value of `ModelType.STRING` is analogous to a Java `Map<String, String>`. A parameter with a `type` of OBJECT and a `value-type` whose value is not of type `ModelType` represents a fully-defined complex object, with the object's legal fields and their values described.
- `expressions-allowed` – boolean – indicates whether the value of the the parameter or return value may be of type `ModelType.EXPRESSION`, instead its standard type (see type and value-type above for discussion of the standard type.) A value of ModelType.EXPRESSION contains a system-property substitution expression that the server will resolve against the server-side system property map before using the value. For example, a parameter named max-threads may have an expression value of `${example.pool.max-threads:10}` instead of just 10. Default value if not present is false.
- `required` – boolean – true if the parameter or return value must have a defined value in the operation or response unless another item included in a list of `alternatives` is defined; false if it may be undefined (implying a null value) even in the absence of alternatives. If not present, true is the default.
- `nillable` – boolean – true if the parameter or return value might not have a defined value in a representation of its portion of the model. A nillable parameter or return value may be undefined either because it is not `required` or because it is required but has `alternatives` and one of the alternatives is defined.
- `default` – the default value for the parameter that will be used in runtime services if the parameter is not explicitly defined and no other parameters listed as `alternatives` are defined.
- `restart-required` – String – One of "no-services", "all-services", "resource-services" or "jvm". Only relevant to attributes whose access-type is read-write. Indicates whether execution of a write-attribute operation whose name parameter specifies this attribute requires a restart of services (or an entire JVM) in order for the change to take effect in the runtime . See discussion of "Applying Updates to Runtime Services" below. Default value is "no-services".
- `alternatives` – List of string – Indicates an exclusive relationship between parameters. If this attribute is defined, the other parameters listed in this descriptor's value should be undefined, even if their required descriptor says true; i.e. the presence of this parameter satisfies the requirement. Note that an parameer that is not explicitly configured but has a `default` value is still regarded as not being defined for purposes of checking whether the exclusive relationship has been violated. Default is undefined; i.e. this does not apply to most parameters.
- `requires` – List of string – Indicates that if this parameter has a value (other than undefined), the other parameters listed in this descriptor's value must also have a value, even if their required descriptor says false. This would typically be used in conjunction with alternatives. For example, parameters "a" and "b" are required, but are alternatives to each other; "c" and "d" are optional. But "b" requires "c" and "d", so if "b" is used, "c" and "d" must also be defined. Default is undefined; i.e. this does not apply to most parameters.

- arbitrary key/value pairs that further describe the attribute value, e.g. "max" =>2. See "Arbitrary Descriptors" below.

Arbitrary Descriptors

The description of an attribute, operation parameter or operation return value type can include arbitrary key/value pairs that provide extra information. Whether a particular key/value pair is present depends on the context, e.g. a pair with key "max" would probably only occur as part of the description of some numeric type.

Following are standard keys and their expected value type. If descriptor authors want to add an arbitrary key/value pair to some descriptor and the semantic matches the meaning of one of the following items, the standard key/value type must be used.

- `min` – int – the minimum value of some numeric type. The absence of this item implies there is no minimum value.
- `max` – int – the maximum value of some numeric type. The absence of this item implies there is no maximum value.
- `min-length` – int – the minimum length of some string, list or byte[] type. The absence of this item implies a minimum length of zero.
- `max-length` – int – the maximum length of some string, list or byte[]. The absence of this item implies there is no maximum value.
- `allowed` – List – a list of legal values. The type of the elements in the list should match the type of the attribute.
- `unit` - The unit of the value, if one is applicable - e.g. ns, ms, s, m, h, KB, MB, TB. See the `org.jboss.as.controller.client.helpers.MeasurementUnit` in the org.jboss.as:jboss-as-controller-client artifact for a listing of legal measurement units..

Some examples:

```
{
     "operation-name" => "incrementFoo",
     "description" => "Increase the value of the 'foo' attribute by the given amount",
     "request-properties" => {
          "increment" => {
               "type" => INT,
               "description" => "The amount to increment",
               "required" => true
     }},
     "reply-properties" => {
               "type" => INT,
               "description" => "The new value",
     }
}
```

```
{
     "operation-name" => "start",
     "description" => "Starts the thing",
     "request-properties" => {},
     "reply-properties" => {}
}
```

Description of Parent/Child Relationships

The address used to target an addressable portion of the model must be an ordered list of key value pairs. The effect of this requirement is the addressable portions of the model naturally form a tree structure, with parent nodes in the tree defining what the valid keys are and the children defining what the valid values are. The parent node also defines the cardinality of the relationship. The description of the parent node includes a children element that describes these relationships:

```
{
     ....
     "children" => {
          "connector" => {
               .... description of the relationship with children of type "connector"
          },
          "virtual-host" => {
               .... description of the relationship with children of type "virtual-host"
          }
}
```

The description of each relationship will include the following elements:

- `description` – String – text description of the relationship
- `model-description` – either "undefined" or a complex structure – This is a node of ModelType.OBJECT, the keys of which are legal values for the value portion of the address of a resource of this type, with the special character '*' indicating the value portion can have an arbitrary value. The values in the node are the full description of the particular child resource (its text description, attributes, operations, children) as detailed above. This `model-description` may also be "undefined", i.e. a null value, if the query that asked for the parent node's description did not include the "recursive" param set to true.

Example with if the recursive flag was set to true:

```
{
     "description" => "The connectors used to handle client connections",
     "model-description" => {
          "*" => {
              "description" => "Handles client connections",
              "min-occurs" => 1,
              "attributes => {
                   ... details of children as documented above
              },
              "operations" => {
                   .... details of operations as documented above
              },
              "children" => {
                   .... details of the children's children
              }
          }
     }
}
```

If the recursive flag was false:

```
{
     "description" => "The connectors used to handle client connections",
     "model-description" => undefined
}
```

Applying Updates to Runtime Services

An attribute or operation description may include a "`restart-required`" descriptor; this section is an explanation of the meaning of that descriptor.

An operation that changes a management resource's persistent configuration usually can also also affect a runtime service associated with the resource. For example, there is a runtime service associated with any host.xml or standalone.xml `<interface>` element; other services in the runtime depend on that service to provide the `InetAddress` associated with the interface. In many cases, an update to a resource's persistent configuration can be immediately applied to the associated runtime service. The runtime service's state is updated to reflect the new value(s).

However, in many cases the runtime service's state cannot be updated without restarting the service. Restarting a service can have broad effects. A restart of a service A will trigger a restart of other services B, C and D that depend A, triggering a restart of services that depend on B, C and D, etc. Those service restarts may very well disrupt handling of end-user requests.

Because restarting a service can be disruptive to end-user request handling, the handlers for management operations will not restart any service without some form of explicit instruction from the end user indicating a service restart is desired. In a few cases, simply executing the operation is an indication the user wants services to restart (e.g. a `/host=master/server-config=server-one:restart` operation in a managed domain, or a `/:reload` operation on a standalone server.) For all other cases, if an operation (or attribute write) cannot be performed without restarting a service, the metadata describing the operation or attribute will include a "`restart-required`" descriptor whose value indicates what is necessary for the operation to affect the runtime:

- `no-services` – Applying the operation to the runtime does not require the restart of any services. This value is the default if the restart-required descriptor is not present.
- `all-services` – The operation can only immediately update the persistent configuration; applying the operation to the runtime will require a subsequent restart of all services in the affected VM. Executing the operation will put the server into a "`reload-required`" state. Until a restart of all services is performed the response to this operation and to any subsequent operation will include a response header "`process-state" => "reload-required`". For a standalone server, a restart of all services can be accomplished by executing the `/:reload` CLI command. For a server in a managed domain, restarting all services currently requires a full restart of the affected server VM (e.g. `/host=master/server-config=server-one:restart`).
- `jvm` --The operation can only immediately update the persistent configuration; applying the operation to the runtime will require a full process restart (i.e. stop the JVM and launch a new JVM). Executing the operation will put the server into a "`restart-required`" state. Until a restart is performed the response to this operation and to any subsequent operation will include a response header "`process-state" => "restart-required`". For a standalone server, a full process restart requires first stopping the server via OS-level operations (Ctrl-C, kill) or via the `/:shutdown` CLI command, and then starting the server again from the command line. For a server in a managed domain, restarting a server requires executing the `/host=<host>/server-config=<server>:restart` operation.
- `resource-services` – The operation can only immediately update the persistent configuration; applying the operation to the runtime will require a subsequent restart of some services associated with the resource. If the operation includes the request header `"allow-resource-service-restart" => true`, the handler for the operation will go ahead and restart the runtime service. Otherwise executing the operation will put the server into a "`reload-required`" state. (See the discussion of "`all-services`" above for more on the "`reload-required`" state.)

5.10.4 The native management API

A standalone WildFly process, or a managed domain Domain Controller or slave Host Controller process can be configured to listen for remote management requests using its "native management interface":

```
<native-interface interface="management" port="9999" security-realm="ManagementRealm"/>
```

(See standalone/configuration/standalone.xml or domain/configuration/host.xml)

The CLI tool that comes with the application server uses this interface, and user can develop custom clients that use it as well. In this section we'll cover the basics on how to develop such a client. We'll also cover details on the format of low-level management operation requests and responses – information that should prove useful for users of the CLI tool as well.

Native Management Client Dependencies

The native management interface uses an open protocol based on the JBoss Remoting library. JBoss Remoting is used to establish a communication channel from the client to the process being managed. Once the communication channel is established the primary traffic over the channel is management requests initiated by the client and asynchronous responses from the target process.

A custom Java-based client should have the maven artifact `org.jboss.as:jboss-as-controller-client` and its dependencies on the classpath. The other dependencies are:

Maven Artifact	Purpose
`org.jboss.remoting:jboss-remoting`	Remote communication
`org.jboss:jboss-dmr`	Detyped representation of the management model
`org.jboss.as:jboss-as-protocol`	Wire protocol for remote WildFly management
`org.jboss.sasl:jboss-sasl`	SASL authentication
`org.jboss.xnio:xnio-api`	Non-blocking IO
`org.jboss.xnio:xnio-nio`	Non-blocking IO
`org.jboss.logging:jboss-logging`	Logging
`org.jboss.threads:jboss-threads`	Thread management
`org.jboss.marshalling:jboss-marshalling`	Marshalling and unmarshalling data to/from streams

The client API is entirely within the `org.jboss.as:jboss-as-controller-client` artifact; the other dependencies are part of the internal implementation of `org.jboss.as:jboss-as-controller-client` and are not compile-time dependencies of any custom client based on it.

The management protocol is an open protocol, so a completely custom client could be developed without using these libraries (e.g. using Python or some other language.)

Working with a ModelControllerClient

The `org.jboss.as.controller.client.ModelControllerClient` class is the main class a custom client would use to manage a WildFly server instance or a Domain Controller or slave Host Controller.

The custom client must have maven artifact `org.jboss.as:jboss-as-controller-client` and its dependencies on the classpath.

Creating the ModelControllerClient

To create a management client that can connect to your target process's native management socket, simply:

```
ModelControllerClient client =
ModelControllerClient.Factory.create(InetAddress.getByName("localhost"), 9999);
```

The address and port are what is configured in the target process' `<management><management-interfaces><native-interface.../>` element.

Typically, however, the native management interface will be secured, requiring clients to authenticate. On the client side, the custom client will need to provide the user's authentication credentials, obtained in whatever manner is appropriate for the client (e.g. from a dialog box in a GUI-based client.) Access to these credentials is provided by passing in an implementation of the `javax.security.auth.callback.CallbackHandler` interface. For example:

```
static ModelControllerClient createClient(final InetAddress host, final int port,
                  final String username, final char[] password, final String securityRealmName)
{

    final CallbackHandler callbackHandler = new CallbackHandler() {

        public void handle(Callback[] callbacks) throws IOException,
UnsupportedCallbackException {
            for (Callback current : callbacks) {
                if (current instanceof NameCallback) {
                    NameCallback ncb = (NameCallback) current;
                    ncb.setName(username);
                } else if (current instanceof PasswordCallback) {
                    PasswordCallback pcb = (PasswordCallback) current;
                    pcb.setPassword(password.toCharArray());
                } else if (current instanceof RealmCallback) {
                    RealmCallback rcb = (RealmCallback) current;
                    rcb.setText(rcb.getDefaultText());
                } else {
                    throw new UnsupportedCallbackException(current);
                }
            }
        }
    };

    return ModelControllerClient.Factory.create(host, port, callbackHandler);
}
```

Creating an operation request object

Management requests are formulated using the `org.jboss.dmr.ModelNode` class from the `jboss-dmr` library. The `jboss-dmr` library allows the complete WildFly management model to be expressed using a very small number of Java types. See Detyped management and the jboss-dmr library for full details on using this library.

Let's show an example of creating an operation request object that can be used to read the resource description for the web subsystem's HTTP connector:

```
ModelNode op = new ModelNode();
op.get("operation").set("read-resource-description");

ModelNode address = op.get("address");
address.add("subsystem", "web");
address.add("connector", "http");

op.get("recursive").set(true);
op.get("operations").set(true);
```

What we've done here is created a ModelNode of type `ModelType.OBJECT` with the following fields:

- `operation` – the name of the operation to invoke. All operation requests **must** include this field and its value must be a String.
- `address` – the address of the resource to invoke the operation against. This field's must be of `ModelType.LIST` with each element in the list being a `ModelType.PROPERTY`. If this field is omitted the operation will target the root resource. The operation can be targeted at any address in the management model; here we are targeting it at the resource for the web subsystem's http connector.

In this case, the request includes two optional parameters:

- `recursive` – true means you want the description of child resources under this resource. Default is false
- `operations` – true means you want the description of operations exposed by the resource to be included. Default is false.

Different operations take different parameters, and some take no parameters at all.

See Format of a Detyped Operation Request for full details on the structure of a ModelNode that will represent an operation request.

The example above produces an operation request ModelNode equivalent to what the CLI produces internally when it parses and executes the following low-level CLI command:

```
[localhost:9999 /]
/subsystem=web/connector=http:read-resource-description(recursive=true,operations=true)
```

Execute the operation and manipulate the result:

The `execute` method sends the operation request ModelNode to the process being managed and returns a ModelNode the contains the process' response:

```
ModelNode returnVal = client.execute(op);
System.out.println(returnVal.get("result").toString());
```

See Format of a Detyped Operation Response for general details on the structure of the "returnVal" ModelNode.

The `execute` operation shown above will block the calling thread until the response is received from the process being managed. `ModelControllerClient` also exposes and API allowing asynchronous invocation:

```
Future<ModelNode> future = client.executeAsync(op);
. . .  // do other stuff
ModelNode returnVal = future.get();
System.out.println(returnVal.get("result").toString());
```

Close the ModelControllerClient

A `ModelControllerClient` can be reused for multiple requests. Creating a new `ModelControllerClient` for each request is an anti-pattern. However, when the `ModelControllerClient` is no longer needed, it should always be explicitly closed, allowing it to close down any connections to the process it was managing and release other resources:

```
client.close();
```

Format of a Detyped Operation Request

The basic method a user of the WildFly 8 programmatic management API would use is very simple:

```
ModelNode execute(ModelNode operation) throws IOException;
```

where the return value is the detyped representation of the response, and `operation` is the detyped representation of the operation being invoked.

The purpose of this section is to document the structure of `operation`.

See Format of a Detyped Operation Response for a discussion of the format of the response.

Simple Operations

A text representation of simple operation would look like this:

```
{
    "operation" => "write-attribute",
    "address" => [
        ("profile" => "production"),
        ("subsystem" => "threads"),
        ("bounded-queue-thread-pool" => "pool1")
    ],
    "name" => "count",
    "value" => 20
}
```

Java code to produce that output would be:

```
ModelNode op = new ModelNode();
op.get("operation").set("write-attribute");
ModelNode addr = op.get("address");
addr.add("profile", "production");
addr.add("subsystem", "threads");
addr.add("bounded-queue-thread-pool", "pool1");
op.get("name").set("count");
op.get("value").set(20);
System.out.println(op);
```

The order in which the outermost elements appear in the request is not relevant. The required elements are:

- `operation` – String – The name of the operation being invoked.
- `address` – the address of the managed resource against which the request should be executed. If not set, the address is the root resource. The address is an ordered list of key-value pairs describing where the resource resides in the overall management resource tree. Management resources are organized in a tree, so the order in which elements in the address occur is important.

The other key/value pairs are parameter names and their values. The names and values should match what is specified in the operation's description.

Parameters may have any name, except for the reserved words `operation`, `address` and `operation-headers`.

Operation Headers

Besides the special operation and address values discussed above, operation requests can also include special "header" values that help control how the operation executes. These headers are created under the special reserved word `operation-headers`:

```
ModelNode op = new ModelNode();
op.get("operation").set("write-attribute");
ModelNode addr = op.get("address");
addr.add("base", "domain");
addr.add("profile", "production");
addr.add("subsystem", "threads");
addr.add("bounded-queue-thread-pool", "pool1");
op.get("name").set("count");
op.get("value").set(20);
op.get("operation-headers", "rollback-on-runtime-failure").set(false);
System.out.println(op);
```

This produces:

```
{
    "operation" => "write-attribute",
    "address" => [
        ("profile" => "production"),
        ("subsystem" => "threads"),
        ("bounded-queue-thread-pool" => "pool1")
    ],
    "name" => "count",
    "value" => 20,
    "operation-headers" => {
        "rollback-on-runtime-failure => false
    }
}
```

The following operation headers are supported:

- `rollback-on-runtime-failure` – boolean, optional, defaults to true. Whether an operation that successfully updates the persistent configuration model should be reverted if it fails to apply to the runtime. Operations that affect the persistent configuration are applied in two stages – first to the configuration model and then to the actual running services. If there is an error applying to the configuration model the operation will be aborted with no configuration change and no change to running services will be attempted. However, operations are allowed to change the configuration model even if there is a failure to apply the change to the running services – if and only if this `rollback-on-runtime-failure` header is set to `false`. So, this header only deals with what happens if there is a problem applying an operation to the running state of a server (e.g. actually increasing the size of a runtime thread pool.)
- `rollout-plan` – only relevant to requests made to a Domain Controller or Host Controller. See "Operations with a Rollout Plan" for details.
- `allow-resource-service-restart` – boolean, optional, defaults to false. Whether an operation that requires restarting some runtime services in order to take effect should do so. See discussion of `resource-services` in the "Applying Updates to Runtime Services" section of the Description of the Management Model section for further details.
- `roles` – String or list of strings. Name(s) of RBAC role(s) the permissions for which should be used when making access control decisions instead of those from the roles normally associated with the user invoking the operation. Only respected if the user is normally associated with a role with all permissions (i.e. SuperUser), meaning this can only be used to reduce permissions for a caller, not to increase permissions.
- `blocking-timeout` – int, optional, defaults to 300. Maximum time, in seconds, that the operation should block at various points waiting for completion. If this period is exceeded, the operation will roll back. Does not represent an overall maximum execution time for an operation; rather it is meant to serve as a sort of fail-safe measure to prevent problematic operations indefinitely tying up resources.

Composite Operations

The root resource for a Domain or Host Controller or an individual server will expose an operation named "`composite`". This operation executes a list of other operations as an atomic unit (although the atomicity requirement can be relaxed. The structure of the request for the "`composite`" operation has the same fundamental structure as a simple operation (i.e. operation name, address, params as key value pairs).

```
{
    "operation" => "composite",
    "address" => [],
    "steps" => [
         {
              "operation" => "write-attribute",
              "address" => [
                   ("profile" => "production"),
                   ("subsystem" => "threads"),
                   ("bounded-queue-thread-pool" => "pool1")
              ],
              "count" => "count",
              "value" => 20
         },
         {
              "operation" => "write-attribute",
              "address" => [
                   ("profile" => "production"),
                   ("subsystem" => "threads"),
                   ("bounded-queue-thread-pool" => "pool2")
              ],
              "name" => "count",
              "value" => 10
         }
    ],
    "operation-headers" => {
        "rollback-on-runtime-failure => false
    }
}
```

The "composite" operation takes a single parameter:

- `steps` – a list, where each item in the list has the same structure as a simple operation request. In the example above each of the two steps is modifying the thread pool configuration for a different pool. There need not be any particular relationship between the steps. Note that the `rollback-on-runtime-failure` and `rollout-plan` operation headers are not supported for the individual steps in a composite operation.

The `rollback-on-runtime-failure` operation header discussed above has a particular meaning when applied to a composite operation, controlling whether steps that successfully execute should be reverted if other steps fail at runtime. Note that if any steps modify the persistent configuration, and any of those steps fail, all steps will be reverted. Partial/incomplete changes to the persistent configuration are not allowed.

Operations with a Rollout Plan

Operations targeted at domain or host level resources can potentially impact multiple servers. Such operations can include a "rollout plan" detailing the sequence in which the operation should be applied to servers as well as policies for detailing whether the operation should be reverted if it fails to execute successfully on some servers.

If the operation includes a rollout plan, the structure is as follows:

```
{
    "operation" => "write-attribute",
    "address" => [
        ("profile" => "production"),
        ("subsystem" => "threads"),
        ("bounded-queue-thread-pool" => "pool1")
    ],
    "name" => "count",
    "value" => 20,
    "operation-headers" => {
        "rollout-plan" => {
            "in-series" => [
                {
                    "concurrent-groups" => {
                        "groupA" => {
                            "rolling-to-servers" => true,
                            "max-failure-percentage" => 20
                        },
                        "groupB" => undefined
                    }
                },
                {
                   "server-group" => {
                        "groupC" => {
                            "rolling-to-servers" => false,
                            "max-failed-servers" => 1
                        }
                    }
                },
                {
                    "concurrent-groups" => {
                        "groupD" => {
                            "rolling-to-servers" => true,
                            "max-failure-percentage" => 20
                        },
                        "groupE" => undefined
                    }
                }
            ],
            "rollback-across-groups" => true
        }
    }
}
```

As you can see, the rollout plan is another structure in the operation-headers section. The root node of the structure allows two children:

- `in-series` – a list – A list of activities that are to be performed in series, with each activity reaching completion before the next step is executed. Each activity involves the application of the operation to the servers in one or more server groups. See below for details on each element in the list.
- `rollback-across-groups` – boolean – indicates whether the need to rollback the operation on all the servers in one server group should trigger a rollback across all the server groups. This is an optional setting, and defaults to `false`.

Each element in the list under the `in-series` node must have one or the other of the following structures:

- `concurrent-groups` – a map of server group names to policies controlling how the operation should be applied to that server group. For each server group in the map, the operation may be applied concurrently. See below for details on the per-server-group policy configuration.
- `server-group` – a single key/value mapping of a server group name to a policy controlling how the operation should be applied to that server group. See below for details on the policy configuration. (Note: there is no difference in plan execution between this and a "`concurrent-groups`" map with a single entry.)

The policy controlling how the operation is applied to the servers within a server group has the following elements, each of which is optional:

- `rolling-to-servers` – boolean – If true, the operation will be applied to each server in the group in series. If false or not specified, the operation will be applied to the servers in the group concurrently.
- `max-failed-servers` – int – Maximum number of servers in the group that can fail to apply the operation before it should be reverted on all servers in the group. The default value if not specified is zero; i.e. failure on any server triggers rollback across the group.
- `max-failure-percentage` – int between 0 and 100 – Maximum percentage of the total number of servers in the group that can fail to apply the operation before it should be reverted on all servers in the group. The default value if not specified is zero; i.e. failure on any server triggers rollback across the group.

If both `max-failed-servers` and `max-failure-percentage` are set, `max-failure-percentage` takes precedence.

Looking at the (contrived) example above, application of the operation to the servers in the domain would be done in 3 phases. If the policy for any server group triggers a rollback of the operation across the server group, all other server groups will be rolled back as well. The 3 phases are:

1. Server groups groupA and groupB will have the operation applied concurrently. The operation will be applied to the servers in groupA in series, while all servers in groupB will handle the operation concurrently. If more than 20% of the servers in groupA fail to apply the operation, it will be rolled back across that group. If any servers in groupB fail to apply the operation it will be rolled back across that group.
2. Once all servers in groupA and groupB are complete, the operation will be applied to the servers in groupC. Those servers will handle the operation concurrently. If more than one server in groupC fails to apply the operation it will be rolled back across that group.
3. Once all servers in groupC are complete, server groups groupD and groupE will have the operation applied concurrently. The operation will be applied to the servers in groupD in series, while all servers in groupE will handle the operation concurrently. If more than 20% of the servers in groupD fail to apply the operation, it will be rolled back across that group. If any servers in groupE fail to apply the operation it will be rolled back across that group.

Default Rollout Plan

All operations that impact multiple servers will be executed with a rollout plan. However, actually specifying the rollout plan in the operation request is not required. If no `rollout-plan` operation header is specified, a default plan will be generated. The plan will have the following characteristics:

- There will only be a single high level phase. All server groups affected by the operation will have the operation applied concurrently.
- Within each server group, the operation will be applied to all servers concurrently.
- Failure on any server in a server group will cause rollback across the group.
- Failure of any server group will result in rollback of all other server groups.

Creating and reusing a Rollout Plan

Since a rollout plan may be quite complex, having to pass it as a header every time can become quickly painful. So instead we can store it in the model and then reference it when we want to use it.
To create a rollout plan you can use the operation `rollout-plan add` like this :

```
rollout-plan add --name=simple --content={"rollout-plan" => {"in-series" => [{"server-group" =>
{"main-server-group" => {"rolling-to-servers" => false,"max-failed-servers" => 1}}},
{"server-group" => {"other-server-group" => {"rolling-to-servers" =>
true,"max-failure-percentage" => 20}}}],"rollback-across-groups" => true}}
```

This will create a rollout plan called `simple` in the content repository.

```
[domain@192.168.1.20:9999 /]
/management-client-content=rollout-plans/rollout-plan=simple:read-resource
{
    "outcome" => "success",
    "result" => {
        "content" => {"rollout-plan" => {
            "in-series" => [
                {"server-group" => {"main-server-group" => {
                    "rolling-to-servers" => false,
                    "max-failed-servers" => 1
                }}},
                {"server-group" => {"other-server-group" => {
                    "rolling-to-servers" => true,
                    "max-failure-percentage" => 20
                }}}
            ],
            "rollback-across-groups" => true
        }},
        "hash" => bytes {
            0x13, 0x12, 0x76, 0x65, 0x8a, 0x28, 0xb8, 0xbc,
            0x34, 0x3c, 0xe9, 0xe6, 0x9f, 0x24, 0x05, 0xd2,
            0x30, 0xff, 0xa4, 0x34
        }
    }
}
```

Now you may reference the roolout plan in your command by adding a header just like this :

```
deploy /quickstart/ejb-in-war/target/wildfly-ejb-in-war.war --all-server-groups
--headers={rollout name=simple}
```

Format of a Detyped Operation Response

As noted previously, the basic method a user of the WildFly 8 programmatic management API would use is very simple:

```
ModelNode execute(ModelNode operation) throws IOException;
```

where the return value is the detyped representation of the response, and `operation` is the detyped representation of the operating being invoked.

The purpose of this section is to document the structure of the return value.

For the format of the request, see Format of a Detyped Operation Request.

Simple Responses

Simple responses are provided by the following types of operations:

- Non-composite operations that target a single server. (See below for more on composite operations).
- Non-composite operations that target a Domain Controller or slave Host Controller and don't require the responder to apply the operation on multiple servers and aggregate their results (e.g. a simple read of a domain configuration property.)

The response will always include a simple boolean outcome field, with one of three possible values:

- `success` – the operation executed successfully
- `failed` – the operation failed
- `cancelled` – the execution of the operation was cancelled. (This would be an unusual outcome for a simple operation which would generally very rapidly reach a point in its execution where it couldn't be cancelled.)

The other fields in the response will depend on whether the operation was successful.

The response for a failed operation:

```
{
    "outcome" => "failed",
    "failure-description" => "[JBAS-12345] Some failure message"
}
```

A response for a successful operation will include an additional field:

- `result` – the return value, or `undefined` for void operations or those that return null

A non-void result:

```
{
    "outcome" => "success",
    "result" => {
        "name" => "Brian",
        "age" => 22
    }
}
```

A void result:

```
{
    "outcome" => "success",
    "result" => undefined
}
```

The response for a cancelled operation has no other fields:

```
{
    "outcome" => "cancelled"
}
```

Response Headers

Besides the standard `outcome`, `result` and `failure-description` fields described above, the response may also include various headers that provide more information about the affect of the operation or about the overall state of the server. The headers will be child element under a field named `response-headers`. For example:

```
{
    "outcome" => "success",
    "result" => undefined,
    "response-headers" => {
        "operation-requires-reload" => true,
        "process-state" => "reload-required"
    }
}
```

A response header is typically related to whether an operation could be applied to the targeted runtime without requiring a restart of some or all services, or even of the target process itself. Please see the "Applying Updates to Runtime Services" section of the Description of the Management Model section for a discussion of the basic concepts related to what happens if an operation requires a service restart to be applied.

The current possible response headers are:

- `operation-requires-reload` – boolean – indicates that the specific operation that has generated this response requires a restart of all services in the process in order to take effect in the runtime. This would typically only have a value of 'true'; the absence of the header is the same as a value of 'false.'
- `operation-requires-restart` – boolean – indicates that the specific operation that has generated this response requires a full process restart in order to take effect in the runtime. This would typically only have a value of 'true'; the absence of the header is the same as a value of 'false.'
- `process-state` – enumeration – Provides information about the overall state of the target process. One of the following values:
  - `starting` – the process is starting
  - `running` – the process is in a normal running state. The `process-state` header would typically not be seen with this value; the absence of the header is the same as a value of 'running'.
  - `reload-required` – some operation (not necessarily this one) has executed that requires a restart of all services in order for a configuration change to take effect in the runtime.
  - `restart-required` – some operation (not necessarily this one) has executed that requires a full process restart in order for a configuration change to take effect in the runtime.
  - `stopping` – the process is stopping

Basic Composite Operation Responses

A composite operation is one that incorporates more than one simple operation in a list and executes them atomically. See the "Composite Operations" section for more information.

Basic composite responses are provided by the following types of operations:

- Composite operations that target a single server.
- Composite operations that target a Domain Controller or a slave Host Controller and don't require the responder to apply the operation on multiple servers and aggregate their results (e.g. a list of simple reads of domain configuration properties.)

The high level format of a basic composite operation response is largely the same as that of a simple operation response, although there is an important semantic difference. For a composite operation, the meaning of the outcome flag is controlled by the value of the operation request's `rollback-on-runtime-failure` header field. If that field was `false` (default is true), the outcome flag will be success if all steps were successfully applied to the persistent configuration even if **none** of the composite operation's steps was successfully applied to the runtime.

What's distinctive about a composite operation response is the `result` field. First, even if the operation was not successful, the `result` field will usually be present. (It won't be present if there was some sort of immediate failure that prevented the responder from even attempting to execute the individual operations.) Second, the content of the `result` field will be a map. Each entry in the map will record the result of an element in the `steps` parameter of the composite operation request. The key for each item in the map will be the string "`step-X`" where "X" is the 1-based index of the step's position in the request's `steps` list. So each individual operation in the composite operation will have its result recorded.

The individual operation results will have the same basic format as the simple operation results described above. However, there are some differences from the simple operation case when the individual operation's `outcome` flag is `failed`. These relate to the fact that in a composite operation, individual operations can be rolled back or not even attempted.

If an individual operation was not even attempted (because the overall operation was cancelled or, more likely, a prior operation failed):

```
{
    "outcome" => "cancelled"
}
```

An individual operation that failed and was rolled back:

```
{
    "outcome" => "failed",
    "failure-description" => "[JBAS-12345] Some failure message",
    "rolled-back" => true
}
```

An individual operation that itself succeeded but was rolled back due to failure of another operation:

```
{
    "outcome" => "failed",
    "result" => {
        "name" => "Brian",
        "age" => 22
    },
    "rolled-back" => true
}
```

An operation that failed and was rolled back:

```
{
    "outcome" => "failed",
    "failure-description" => "[JBAS-12345] Some failure message",
    "rolled-back" => true
}
```

Here's an example of the response for a successful 2 step composite operation:

```
{
    "outcome" => "success",
    "result" => [
        {
            "outcome" => "success",
            "result" => {
                "name" => "Brian",
                "age" => 22
            }
        },
        {
            "outcome" => "success",
            "result" => undefined
        }
    ]
}
```

And for a failed 3 step composite operation, where the first step succeeded and the second failed, triggering cancellation of the 3rd and rollback of the others:

```
{
    "outcome" => "failed",
    "failure-description" => "[JBAS-99999] Composite operation failed; see individual operation
results for details",
    "result" => [
        {
            "outcome" => "failed",
            "result" => {
                "name" => "Brian",
                "age" => 22
            },
            "rolled-back" => true
        },
        {
            "outcome" => "failed",
            "failure-description" => "[JBAS-12345] Some failure message",
            "rolled-back" => true
        },
        {
            "outcome" => "cancelled"
        }
    ]
}
```

Multi-Server Responses

Multi-server responses are provided by operations that target a Domain Controller or slave Host Controller and require the responder to apply the operation on multiple servers and aggregate their results (e.g. nearly all domain or host configuration updates.)

Multi-server operations are executed in several stages.

First, the operation may need to be applied against the authoritative configuration model maintained by the Domain Controller (for `domain.xml` confgurations) or a Host Controller (for a `host.xml` configuration). If there is a failure at this stage, the operation is automatically rolled back, with a response like this:

```
{
    "outcome" => "failed",
    "failure-description" => {
        "domain-failure-description" => "[JBAS-33333] Failed to apply X to the domain model"
    }
}
```

If the operation was addressed to the domain model, in the next stage the Domain Controller will ask each slave Host Controller to apply it to its local copy of the domain model. If any Host Controller fails to do so, the Domain Controller will tell all Host Controllers to revert the change, and it will revert the change locally as well. The response to the client will look like this:

```
{
    "outcome" => "failed",
    "failure-description" => {
        "host-failure-descriptions" => {
            "hostA" => "[DOM-3333] Failed to apply to the domain model",
            "hostB" => "[DOM-3333] Failed to apply to the domain model"
        }
    }
}
```

If the preceding stages succeed, the operation will be pushed to all affected servers. If the operation is successful on all servers, the response will look like this (this example operation has a void response, hence the result for each server is undefined):

```
{
    "outcome" => "success",
    "result" => undefined,
    "server-groups" => {
        "groupA" => {
            "serverA-1" => {
                "host" => "host1",
                "response" => {
                    "outcome" => "success",
                    "result" => undefined
                }
            },
            "serverA-2" => {
                "host" => "host2",
                "response" => {
                    "outcome" => "success",
                    "result" => undefined
                }
            }
        },
        "groupB" => {
            "serverB-1" => {
                "host" => "host1",
                "response" => {
                    "outcome" => "success",
                    "result" => undefined
                }
            },
            "serverB-2" => {
                "host" => "host2",
                "response" => {
                    "outcome" => "success",
                    "result" => undefined
                }
            }
        }
    }
}
```

The operation need not succeed on all servers in order to get an `"outcome" => "success"` result. All that is required is that it succeed on at least one server without the rollback policies in the rollout plan triggering a rollback on that server. An example response in such a situation would look like this:

```
{
    "outcome" => "success",
    "result" => undefined,
    "server-groups" => {
        "groupA" => {
            "serverA-1" => {
                "host" => "host1",
                "response" => {
                    "outcome" => "success",
                    "result" => undefined
                }
            },
            "serverA-2" => {
                "host" => "host2",
                "response" => {
                    "outcome" => "success",
                    "result" => undefined
                }
            }
        },
        "groupB" => {
            "serverB-1" => {
                "host" => "host1",
                "response" => {
                    "outcome" => "success",
                    "result" => undefined,
                    "rolled-back" => true
                }
            },
            "serverB-2" => {
                "host" => "host2",
                "response" => {
                    "outcome" => "success",
                    "result" => undefined,
                    "rolled-back" => true
                }
            },
            "serverB-3" => {
                "host" => "host3",
                "response" => {
                    "outcome" => "failed",
                    "failure-description" => "[DOM-4556] Something didn't work right",
                    "rolled-back" => true
                }
            }
        }
    }
}
```

Finally, if the operation fails or is rolled back on all servers, an example response would look like this:

```
{
    "outcome" => "failed",
    "server-groups" => {
        "groupA" => {
            "serverA-1" => {
                "host" => "host1",
                "response" => {
                    "outcome" => "success",
                    "result" => undefined
                }
            },
            "serverA-2" => {
                "host" => "host2",
                "response" => {
                    "outcome" => "success",
                    "result" => undefined
                }
            }
        },
        "groupB" => {
            "serverB-1" => {
                "host" => "host1",
                "response" => {
                    "outcome" => "failed",
                    "result" => undefined,
                    "rolled-back" => true
                }
            },
            "serverB-2" => {
                "host" => "host2",
                "response" => {
                    "outcome" => "failed",
                    "result" => undefined,
                    "rolled-back" => true
                }
            },
            "serverB-3" => {
                "host" => "host3",
                "response" => {
                    "outcome" => "failed",
                    "failure-description" => "[DOM-4556] Something didn't work right",
                    "rolled-back" => true
                }
            }
        }
    }
}
```

5.11 CLI Recipes

5.11.1 Properties

Adding, reading and removing system property using CLI

For standalone mode:

```
$ ./bin/jboss-cli.sh --connect controller=IP_ADDRESS
[standalone@IP_ADDRESS:9990 /] /system-property=foo:add(value=bar)
[standalone@IP_ADDRESS:9990 /] /system-property=foo:read-resource
{
    "outcome" => "success",
    "result" => {"value" => "bar"}
}
[standalone@IP_ADDRESS:9990 /] /system-property=foo:remove
{"outcome" => "success"}
```

For domain mode the same commands are used, you can add/read/remove system properties for:
All hosts and server instances in domain

```
[domain@IP_ADDRESS:9990 /] /system-property=foo:add(value=bar)
[domain@IP_ADDRESS:9990 /] /system-property=foo:read-resource
[domain@IP_ADDRESS:9990 /] /system-property=foo:remove
```

Host and its server instances

```
[domain@IP_ADDRESS:9990 /] /host=master/system-property=foo:add(value=bar)
[domain@IP_ADDRESS:9990 /] /host=master/system-property=foo:read-resource
[domain@IP_ADDRESS:9990 /] /host=master/system-property=foo:remove
```

Just one server instance

```
[domain@IP_ADDRESS:9990 /]
/host=master/server-config=server-one/system-property=foo:add(value=bar)
[domain@IP_ADDRESS:9990 /]
/host=master/server-config=server-one/system-property=foo:read-resource
[domain@IP_ADDRESS:9990 /] /host=master/server-config=server-one/system-property=foo:remove
```

Overview of all system properties

Overview of all system properties in WildFly including OS system properties and properties specified on command line using -D, -P or --properties arguments.

Standalone

```
[standalone@IP_ADDRESS:9990 /]
/core-service=platform-mbean/type=runtime:read-attribute(name=system-properties)
```

Domain

```
[domain@IP_ADDRESS:9990 /]
/host=master/core-service=platform-mbean/type=runtime:read-attribute(name=system-properties)
[domain@IP_ADDRESS:9990 /]
/host=master/server=server-one/core-service=platform-mbean/type=runtime:read-attribute(name=system
```

5.11.2 Configuration

List Subsystems

```
[standalone@localhost:9990 /] /:read-children-names(child-type=subsystem)
{
    "outcome" => "success",
    "result" => [
        "batch",
        "datasources",
        "deployment-scanner",
        "ee",
        "ejb3",
        "infinispan",
        "io",
        "jaxrs",
        "jca",
        "jdr",
        "jmx",
        "jpa",
        "jsf",
        "logging",
        "mail",
        "naming",
        "pojo",
        "remoting",
        "resource-adapters",
        "sar",
        "security",
        "threads",
        "transactions",
        "undertow",
        "webservices",
        "weld"
    ]
}
```

List description of available attributes and childs

Descriptions, possible attribute type and values, permission and whether expressions (*${ ... }*) are allowed from the underlying model are shown by the read-resource-description command.

```
/subsystem=datasources/data-source=ExampleDS:read-resource-description
{
    "outcome" => "success",
    "result" => {
        "description" => "A JDBC data-source configuration",
        "head-comment-allowed" => true,
        "tail-comment-allowed" => true,
        "attributes" => {
            "connection-url" => {
                "type" => STRING,
                "description" => "The JDBC driver connection URL",
                "expressions-allowed" => true,
                "nillable" => false,
                "min-length" => 1L,
                "max-length" => 2147483647L,
                "access-type" => "read-write",
                "storage" => "configuration",
                "restart-required" => "no-services"
            },
            "driver-class" => {
                "type" => STRING,
                "description" => "The fully qualified name of the JDBC driver class",
                "expressions-allowed" => true,
                "nillable" => true,
                "min-length" => 1L,
                "max-length" => 2147483647L,
                "access-type" => "read-write",
                "storage" => "configuration",
                "restart-required" => "no-services"
            },
            "datasource-class" => {
                "type" => STRING,
                "description" => "The fully qualified name of the JDBC datasource class",
                "expressions-allowed" => true,
                "nillable" => true,
                "min-length" => 1L,
                "max-length" => 2147483647L,
                "access-type" => "read-write",
                "storage" => "configuration",
                "restart-required" => "no-services"
            },
            "jndi-name" => {
                "type" => STRING,
                "description" => "Specifies the JNDI name for the datasource",
                "expressions-allowed" => true,
                "nillable" => false,
                "access-type" => "read-write",
                "storage" => "configuration",
                "restart-required" => "no-services"
            },
           ...
```

View configuration as XML for domain model or host model

Assume you have a host that is called master

```
[domain@localhost:9990 /] /host=master:read-config-as-xml
```

Just for the domain or standalone

```
[domain@localhost:9990 /] :read-config-as-xml
```

Take a snapshot of what the current domain is

```
[domain@localhost:9990 /] :take-snapshot()
{
    "outcome" => "success",
    "result" => {
        "domain-results" => {"step-1" => {"name" =>
"JBOSS_HOME/domain/configuration/domain_xml_history/snapshot/20110908-165222603domain.xml"}},
        "server-operations" => undefined
    }
}
```

Take the latest snapshot of the host.xml for a particular host

Assume you have a host that is called master

```
[domain@localhost:9990 /]  /host=master:take-snapshot
{
    "outcome" => "success",
    "result" => {
        "domain-results" => {"step-1" => {"name" =>
"JBOSS_HOME/domain/configuration/host_xml_history/snapshot/20110908-165640215host.xml"}},
        "server-operations" => undefined
    }
}
```

How to get interface address

The attribute for interface is named "resolved-address". It's a runtime attribute so it does not show up in :read-resource by default. You have to add the "include-runtime" parameter.

```
./jboss-cli.sh --connect
Connected to standalone controller at localhost:9990
[standalone@localhost:9990 /] cd interface=public
[standalone@localhost:9990 interface=public] :read-resource(include-runtime=true)
{
     "outcome" => "success",
     "result" => {
         "any" => undefined,
         "any-address" => undefined,
         "any-ipv4-address" => undefined,
         "any-ipv6-address" => undefined,
         "criteria" => [("inet-address" => expression "${jboss.bind.address:127.0.0.1}")],
         "inet-address" => expression "${jboss.bind.address:127.0.0.1}",
         "link-local-address" => undefined,
         "loopback" => undefined,
         "loopback-address" => undefined,
         "multicast" => undefined,
         "name" => "public",
         "nic" => undefined,
         "nic-match" => undefined,
         "not" => undefined,
         "point-to-point" => undefined,
         "public-address" => undefined,
         "resolved-address" => "127.0.0.1",
         "site-local-address" => undefined,
         "subnet-match" => undefined,
         "up" => undefined,
         "virtual" => undefined
     }
}
[standalone@localhost:9990 interface=public] :read-attribute(name=resolved-address)
{
     "outcome" => "success",
     "result" => "127.0.0.1"
}
```

It's similar for domain, just specify path to server instance:

```
[domain@localhost:9990 /]
/host=master/server=server-one/interface=public:read-attribute(name=resolved-address)
{
    "outcome" => "success",
    "result" => "127.0.0.1"
}
```

5.11.3 Runtime

Get all configuration and runtime details from CLI

```
./bin/jboss-cli.sh -c command=":read-resource(include-runtime=true, recursive=true,
recursive-depth=10)"
```

5.11.4 Scripting

Windows and "Press any key to continue ..." issue

WildFly scripts for Windows end with "Press any key to continue ...". This behavior is useful when script is executed by double clicking the script but not when you need to invoke several commands from custom script (e.g. 'bin/jboss-admin.bat --connect command=:shutdown').

To avoid "Press any key to continue ..." message you need to specify NOPAUSE variable. Call 'set NOPAUSE=true' in command line before running any WildFly 8 .bat script or include it in your custom script before invoking scripts from WildFly.

5.11.5 Statistics

Read statistics of active datasources

```
/subsystem=datasources/data-source=ExampleDS/statistics=pool:read-resource(include-runtime=true)
/subsystem=datasources/data-source=ExampleDS/statistics=jdbc:read-resource(include-runtime=true)
```

or

```
/subsystem=datasources/data-source=ExampleDS:read-resource(include-runtime=true,recursive=true)
```

5.11.6 Deployment

Undeploying and redeploying multiple deployments

CLI offers a way to efficiently undeploy or redeploy deployments in one simple command.

- To disable all enabled deployments: *undeploy --keep-content* *
- To redeploy all disabled deployments: *deploy --name*=*

Incremental deployment with the CLI

It can be desirable to incrementally create and(or) update a WildFly deployment. This chapter details how this can be achieved using the WildFly CLI tool.

Steps to create an empty deployment and add an index html file.

1. Create an empty deployment named my app:

```
[standalone@localhost:9990 /] /deployment=myapp:add(content=[{empty=true}])
```

2. Add an index.html to my app:

```
[standalone@localhost:9990 /]
/deployment=myapp:add-content(content=[{input-stream-index=<press TAB>
```

   Then use completion to navigate to your index.html file.
3. Provide a target name for index.html inside the deployment and execute the operation:

```
[standalone@localhost:9990 /]
/deployment=myapp:add-content(content=[{input-stream-index=./index.html,
target-path=index.xhtml}]
```

4. Your content has been added, you can browse the content of a deployment using the browse-content operation:

```
[standalone@localhost:9990 /] /deployment=myapp:browse-content(path=./)
```

5. You can display (or save) the content of a deployed file using the *attachement* command:

```
attachment display --operation=/deployment=myapp:read-content(path=index.xhtml)
```

6. You can remove content from a deployment:

```
/deployment=myapp:remove-content(paths=[./index.xhtml])
```

Tips

- *add-content* operation allows you to add more than one file (*content* argument is a list of complex types).
- CLI offers completion for *browse-content*'s *path* and *remove-content*'s *paths* argument.
- You can safely use operations that are using attached streams in batch operations. In the case of batch operations, streams are attached to the composite operation.

On Windows, path separator '\' needs to be escaped, this is a limitation of CLI handling complex types. The file path completion is automatically escaping the paths it is proposing.

Notes for server side operation Handler implementors

In order to benefit from CLI support for attached file streams and file system completion, you need to properly structure your operation arguments. Steps to create an operation that receives a list of file streams attached to the operation:

1. Define your operation argument as a *LIST* of *INT* (The *LIST value-type* must be of type *INT*).
2. In the description of your argument, add the 2 following boolean descriptors: *filesystem-path* and *attached-streams*

When your operation is called from the CLI, file system completion will be automatically proposed for your argument. At execution time, the file system paths will be automatically converted onto the index of the attached streams.

5.11.7 Downloading files with the CLI

Some management resources are exposing the content of files in the matter of *streams*. Streams returned by a management operation are attached to the headers of the management response. The CLI command *attachment* (see CLI help for a detailed description of this command) allows to display or save the content of the attached streams.

- Displaying the content of server.log file:

```
attachment display
--operation=/subsystem=logging/log-file=server.log:read-resource(include-runtime)
```

- Saving locally the server.log file:

```
attachment save
--operation=/subsystem=logging/log-file=server.log:read-resource(include-runtime)
--file=./server.log
```

- Displaying the content of a deployed file:

```
attachment display --operation=/deployment=myapp:read-content(path=index.xhtml)
```

- By default existing files will be preserved. Use the option *--overwrite* to overwrite existing file.
- *attachment* can be used in batch mode.

5.12 All WildFly documentation

There are several guides in the WildFly documentation series. This list gives an overview of each of the guides:

*Getting Started Guide - Explains how to download and start WildFly.
*Getting Started Developing Applications Guide - Talks you through developing your first applications on WildFly, and introduces you to JBoss Tools and how to deploy your applications.
*JavaEE 6 Tutorial - A Java EE 6 Tutorial.
*Admin Guide - Tells you how to configure and manage your WildFly instances.
*Developer Guide - Contains concepts that you need to be aware of when developing applications for WildFly. Classloading is explained in depth.
*High Availability Guide - Reference guide for how to set up clustered WildFly instances.
*Extending WildFly - A guide to adding new functionality to WildFly.

5.13 CLI Recipes

5.13.1 Properties

Adding, reading and removing system property using CLI

For standalone mode:

```
$ ./bin/jboss-cli.sh --connect controller=IP_ADDRESS
[standalone@IP_ADDRESS:9990 /] /system-property=foo:add(value=bar)
[standalone@IP_ADDRESS:9990 /] /system-property=foo:read-resource
{
    "outcome" => "success",
    "result" => {"value" => "bar"}
}
[standalone@IP_ADDRESS:9990 /] /system-property=foo:remove
{"outcome" => "success"}
```

For domain mode the same commands are used, you can add/read/remove system properties for:
All hosts and server instances in domain

```
[domain@IP_ADDRESS:9990 /] /system-property=foo:add(value=bar)
[domain@IP_ADDRESS:9990 /] /system-property=foo:read-resource
[domain@IP_ADDRESS:9990 /] /system-property=foo:remove
```

Host and its server instances

```
[domain@IP_ADDRESS:9990 /] /host=master/system-property=foo:add(value=bar)
[domain@IP_ADDRESS:9990 /] /host=master/system-property=foo:read-resource
[domain@IP_ADDRESS:9990 /] /host=master/system-property=foo:remove
```

Just one server instance

```
[domain@IP_ADDRESS:9990 /]
/host=master/server-config=server-one/system-property=foo:add(value=bar)
[domain@IP_ADDRESS:9990 /]
/host=master/server-config=server-one/system-property=foo:read-resource
[domain@IP_ADDRESS:9990 /] /host=master/server-config=server-one/system-property=foo:remove
```

Overview of all system properties

Overview of all system properties in WildFly including OS system properties and properties specified on command line using -D, -P or --properties arguments.

Standalone

```
[standalone@IP_ADDRESS:9990 /]
/core-service=platform-mbean/type=runtime:read-attribute(name=system-properties)
```

Domain

```
[domain@IP_ADDRESS:9990 /]
/host=master/core-service=platform-mbean/type=runtime:read-attribute(name=system-properties)
[domain@IP_ADDRESS:9990 /]
/host=master/server=server-one/core-service=platform-mbean/type=runtime:read-attribute(name=system
```

5.13.2 Configuration

List Subsystems

```
[standalone@localhost:9990 /] /:read-children-names(child-type=subsystem)
{
    "outcome" => "success",
    "result" => [
        "batch",
        "datasources",
        "deployment-scanner",
        "ee",
        "ejb3",
        "infinispan",
        "io",
        "jaxrs",
        "jca",
        "jdr",
        "jmx",
        "jpa",
        "jsf",
        "logging",
        "mail",
        "naming",
        "pojo",
        "remoting",
        "resource-adapters",
        "sar",
        "security",
        "threads",
        "transactions",
        "undertow",
        "webservices",
        "weld"
    ]
}
```

List description of available attributes and childs

Descriptions, possible attribute type and values, permission and whether expressions (*${ ... }*) are allowed from the underlying model are shown by the read-resource-description command.

```
/subsystem=datasources/data-source=ExampleDS:read-resource-description
{
    "outcome" => "success",
    "result" => {
        "description" => "A JDBC data-source configuration",
        "head-comment-allowed" => true,
        "tail-comment-allowed" => true,
        "attributes" => {
            "connection-url" => {
                "type" => STRING,
                "description" => "The JDBC driver connection URL",
                "expressions-allowed" => true,
                "nillable" => false,
                "min-length" => 1L,
                "max-length" => 2147483647L,
                "access-type" => "read-write",
                "storage" => "configuration",
                "restart-required" => "no-services"
            },
            "driver-class" => {
                "type" => STRING,
                "description" => "The fully qualified name of the JDBC driver class",
                "expressions-allowed" => true,
                "nillable" => true,
                "min-length" => 1L,
                "max-length" => 2147483647L,
                "access-type" => "read-write",
                "storage" => "configuration",
                "restart-required" => "no-services"
            },
            "datasource-class" => {
                "type" => STRING,
                "description" => "The fully qualified name of the JDBC datasource class",
                "expressions-allowed" => true,
                "nillable" => true,
                "min-length" => 1L,
                "max-length" => 2147483647L,
                "access-type" => "read-write",
                "storage" => "configuration",
                "restart-required" => "no-services"
            },
            "jndi-name" => {
                "type" => STRING,
                "description" => "Specifies the JNDI name for the datasource",
                "expressions-allowed" => true,
                "nillable" => false,
                "access-type" => "read-write",
                "storage" => "configuration",
                "restart-required" => "no-services"
            },
           ...
```

View configuration as XML for domain model or host model

Assume you have a host that is called master

```
[domain@localhost:9990 /] /host=master:read-config-as-xml
```

Just for the domain or standalone

```
[domain@localhost:9990 /] :read-config-as-xml
```

Take a snapshot of what the current domain is

```
[domain@localhost:9990 /] :take-snapshot()
{
    "outcome" => "success",
    "result" => {
        "domain-results" => {"step-1" => {"name" =>
"JBOSS_HOME/domain/configuration/domain_xml_history/snapshot/20110908-165222603domain.xml"}},
        "server-operations" => undefined
    }
}
```

Take the latest snapshot of the host.xml for a particular host

Assume you have a host that is called master

```
[domain@localhost:9990 /]  /host=master:take-snapshot
{
    "outcome" => "success",
    "result" => {
        "domain-results" => {"step-1" => {"name" =>
"JBOSS_HOME/domain/configuration/host_xml_history/snapshot/20110908-165640215host.xml"}},
        "server-operations" => undefined
    }
}
```

How to get interface address

The attribute for interface is named "resolved-address". It's a runtime attribute so it does not show up in :read-resource by default. You have to add the "include-runtime" parameter.

```
./jboss-cli.sh --connect
Connected to standalone controller at localhost:9990
[standalone@localhost:9990 /] cd interface=public
[standalone@localhost:9990 interface=public] :read-resource(include-runtime=true)
{
     "outcome" => "success",
     "result" => {
         "any" => undefined,
         "any-address" => undefined,
         "any-ipv4-address" => undefined,
         "any-ipv6-address" => undefined,
         "criteria" => [("inet-address" => expression "${jboss.bind.address:127.0.0.1}")],
         "inet-address" => expression "${jboss.bind.address:127.0.0.1}",
         "link-local-address" => undefined,
         "loopback" => undefined,
         "loopback-address" => undefined,
         "multicast" => undefined,
         "name" => "public",
         "nic" => undefined,
         "nic-match" => undefined,
         "not" => undefined,
         "point-to-point" => undefined,
         "public-address" => undefined,
         "resolved-address" => "127.0.0.1",
         "site-local-address" => undefined,
         "subnet-match" => undefined,
         "up" => undefined,
         "virtual" => undefined
     }
}
[standalone@localhost:9990 interface=public] :read-attribute(name=resolved-address)
{
     "outcome" => "success",
     "result" => "127.0.0.1"
}
```

It's similar for domain, just specify path to server instance:

```
[domain@localhost:9990 /]
/host=master/server=server-one/interface=public:read-attribute(name=resolved-address)
{
    "outcome" => "success",
    "result" => "127.0.0.1"
}
```

5.13.3 Runtime

Get all configuration and runtime details from CLI

```
./bin/jboss-cli.sh -c command=":read-resource(include-runtime=true, recursive=true,
recursive-depth=10)"
```

5.13.4 Scripting

Windows and "Press any key to continue ..." issue

WildFly scripts for Windows end with "Press any key to continue ...". This behavior is useful when script is executed by double clicking the script but not when you need to invoke several commands from custom script (e.g. 'bin/jboss-admin.bat --connect command=:shutdown').

To avoid "Press any key to continue ..." message you need to specify NOPAUSE variable. Call 'set NOPAUSE=true' in command line before running any WildFly 8 .bat script or include it in your custom script before invoking scripts from WildFly.

5.13.5 Statistics

Read statistics of active datasources

```
/subsystem=datasources/data-source=ExampleDS/statistics=pool:read-resource(include-runtime=true)
/subsystem=datasources/data-source=ExampleDS/statistics=jdbc:read-resource(include-runtime=true)
```

or

```
/subsystem=datasources/data-source=ExampleDS:read-resource(include-runtime=true,recursive=true)
```

5.13.6 Deployment

Undeploying and redeploying multiple deployments

CLI offers a way to efficiently undeploy or redeploy deployments in one simple command.

- To disable all enabled deployments: *undeploy --keep-content* *
- To redeploy all disabled deployments: *deploy --name*=*

Incremental deployment with the CLI

It can be desirable to incrementally create and(or) update a WildFly deployment. This chapter details how this can be achieved using the WildFly CLI tool.

Steps to create an empty deployment and add an index html file.

1. Create an empty deployment named my app:

```
[standalone@localhost:9990 /] /deployment=myapp:add(content=[{empty=true}])
```

2. Add an index.html to my app:

```
[standalone@localhost:9990 /]
/deployment=myapp:add-content(content=[{input-stream-index=<press TAB>
```

   Then use completion to navigate to your index.html file.
3. Provide a target name for index.html inside the deployment and execute the operation:

```
[standalone@localhost:9990 /]
/deployment=myapp:add-content(content=[{input-stream-index=./index.html,
target-path=index.xhtml}]
```

4. Your content has been added, you can browse the content of a deployment using the browse-content operation:

```
[standalone@localhost:9990 /] /deployment=myapp:browse-content(path=./)
```

5. You can display (or save) the content of a deployed file using the *attachement* command:

```
attachment display --operation=/deployment=myapp:read-content(path=index.xhtml)
```

6. You can remove content from a deployment:

```
/deployment=myapp:remove-content(paths=[./index.xhtml])
```

Tips

- *add-content* operation allows you to add more than one file (*content* argument is a list of complex types).
- CLI offers completion for *browse-content*'s *path* and *remove-content*'s *paths* argument.
- You can safely use operations that are using attached streams in batch operations. In the case of batch operations, streams are attached to the composite operation.

On Windows, path separator '\' needs to be escaped, this is a limitation of CLI handling complex types. The file path completion is automatically escaping the paths it is proposing.

Notes for server side operation Handler implementors

In order to benefit from CLI support for attached file streams and file system completion, you need to properly structure your operation arguments. Steps to create an operation that receives a list of file streams attached to the operation:

1. Define your operation argument as a *LIST* of *INT* (The *LIST value-type* must be of type *INT*).
2. In the description of your argument, add the 2 following boolean descriptors: *filesystem-path* and *attached-streams*

When your operation is called from the CLI, file system completion will be automatically proposed for your argument. At execution time, the file system paths will be automatically converted onto the index of the attached streams.

5.13.7 Downloading files with the CLI

Some management resources are exposing the content of files in the matter of *streams*. Streams returned by a management operation are attached to the headers of the management response. The CLI command *attachment* (see CLI help for a detailed description of this command) allows to display or save the content of the attached streams.

- Displaying the content of server.log file:

```
attachment display
--operation=/subsystem=logging/log-file=server.log:read-resource(include-runtime)
```

- Saving locally the server.log file:

```
attachment save
--operation=/subsystem=logging/log-file=server.log:read-resource(include-runtime)
--file=./server.log
```

- Displaying the content of a deployed file:

```
attachment display --operation=/deployment=myapp:read-content(path=index.xhtml)
```

- By default existing files will be preserved. Use the option *--overwrite* to overwrite existing file.
- *attachment* can be used in batch mode.

5.14 Core management concepts

5.14.1 Operating modes

WildFly can be booted in two different modes. A *managed domain* allows you to run and manage a multi-server topology. Alternatively, you can run a *standalone server* instance.

Standalone Server

For many use cases, the centralized management capability available via a managed domain is not necessary. For these use cases, a WildFly instance can be run as a "standalone server". A standalone server instance is an independent process, much like an JBoss Application Server 3, 4, 5, or 6 instance is. Standalone instances can be launched via the `standalone.sh` or `standalone.bat` launch scripts.

If more than one standalone instance is launched and multi-server management is desired, it is the user's responsibility to coordinate management across the servers. For example, to deploy an application across all of the standalone servers, the user would need to individually deploy the application on each server.

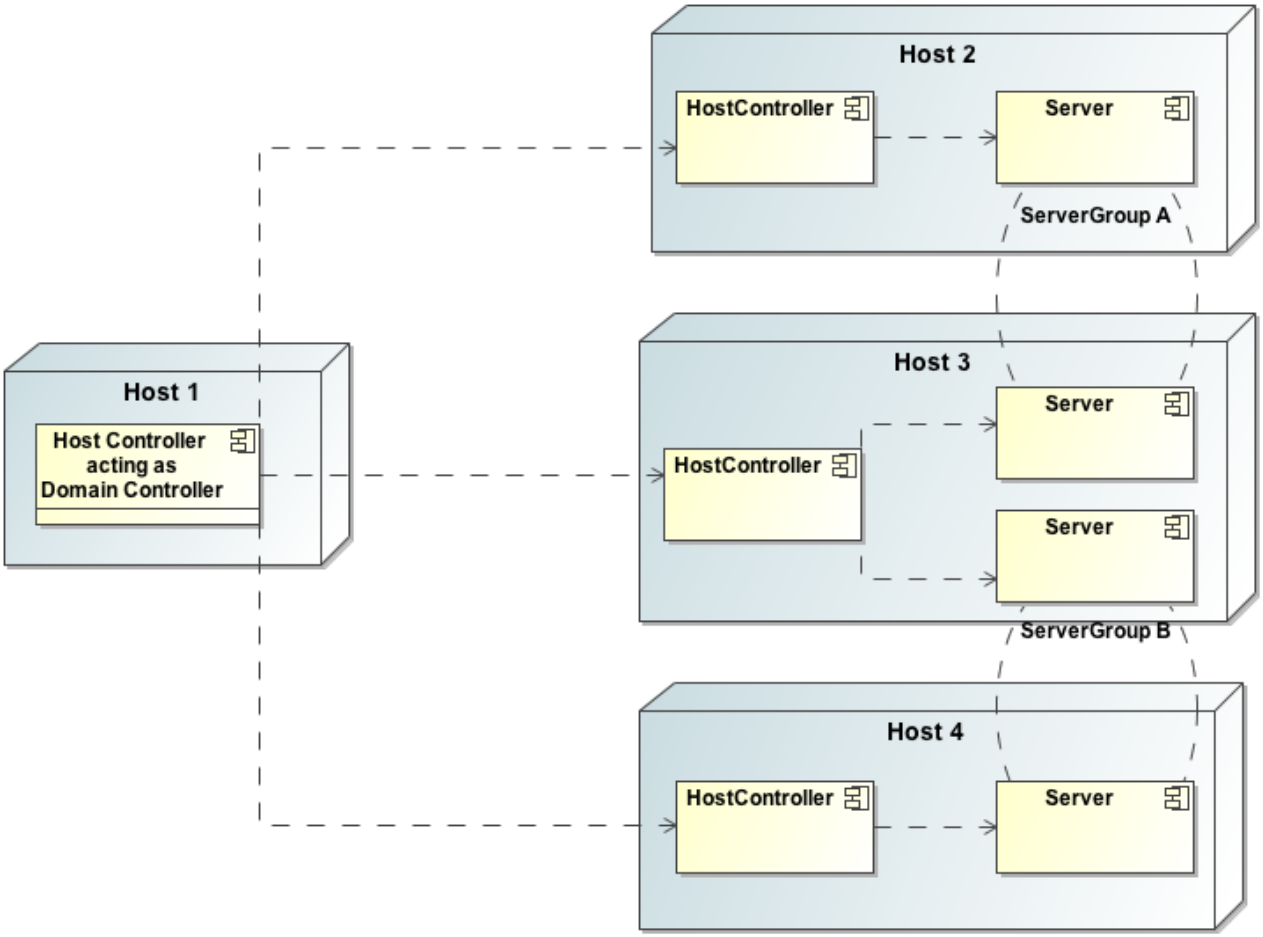
It is perfectly possible to launch multiple standalone server instances and have them form an HA cluster, just like it was possible with JBoss Application Server 3, 4, 5 and 6.

Managed Domain

One of the primary new features of WildFly is the ability to manage multiple WildFly instances from a single control point. A collection of such servers is referred to as the members of a "domain" with a single Domain Controller process acting as the central management control point. All of the WildFly instances in the domain share a common management policy, with the Domain Controller acting to ensure that each server is configured according to that policy. Domains can span multiple physical (or virtual) machines, with all WildFly instances on a given host under the control of a special Host Controller process. One Host Controller instance is configured to act as the central Domain Controller. The Host Controller on each host interacts with the Domain Controller to control the lifecycle of the application server instances running on its host and to assist the Domain Controller in managing them.

When you launch a WildFly managed domain on a host (via the `domain.sh` or `domain.bat` launch scripts) your intent is to launch a Host Controller and usually at least one WildFly instance. On one of the hosts the Host Controller should be configured to act as the Domain Controller. See Domain Setup for details.

The following is an example managed domain topology:



Host

Each "Host" box in the above diagram represents a physical or virtual host. A physical host can contain zero, one or more server instances.

Host Controller

When the domain.sh or domain.bat script is run on a host, a process known as a Host Controller is launched. The Host Controller is solely concerned with server management; it does not itself handle application server workloads. The Host Controller is responsible for starting and stopping the individual application server processes that run on its host, and interacts with the Domain Controller to help manage them.

Each Host Controller by default reads its configuration from the `domain/configuration/host.xml` file located in the unzipped WildFly installation on its host's filesystem. The `host.xml` file contains configuration information that is specific to the particular host. Primarily:

- the listing of the names of the actual WildFly instances that are meant to run off of this installation.
- configuration of how the Host Controller is to contact the Domain Controller to register itself and access the domain configuration. This may either be configuration of how to find and contact a remote Domain Controller, or a configuration telling the Host Controller to itself act as the Domain Controller.
- configuration of items that are specific to the local physical installation. For example, named interface definitions declared in `domain.xml` (see below) can be mapped to an actual machine-specific IP address in `host.xml`. Abstract path names in `domain.xml` can be mapped to actual filesystem paths in `host.xml`.

Domain Controller

One Host Controller instance is configured to act as the central management point for the entire domain, i.e. to be the Domain Controller. The primary responsibility of the Domain Controller is to maintain the domain's central management policy, to ensure all Host Controllers are aware of its current contents, and to assist the Host Controllers in ensuring any running application server instances are configured in accordance with this policy. This central management policy is stored by default in the `domain/configuration/domain.xml` file in the unzipped WildFly installation on Domain Controller's host's filesystem.

A `domain.xml` file must be located in the `domain/configuration` directory of an installation that's meant to run the Domain Controller. It does not need to be present in installations that are not meant to run a Domain Controller; i.e. those whose Host Controller is configured to contact a remote Domain Controller. The presence of a `domain.xml` file on such a server does no harm.

The `domain.xml` file includes, among other things, the configuration of the various "profiles" that WildFly instances in the domain can be configured to run. A profile configuration includes the detailed configuration of the various subsystems that comprise that profile (e.g. an embedded JBoss Web instance is a subsystem; a JBoss TS transaction manager is a subsystem, etc). The domain configuration also includes the definition of groups of sockets that those subsystems may open. The domain configuration also includes the definition of "server groups":

Server Group

A server group is set of server instances that will be managed and configured as one. In a managed domain each application server instance is a member of a server group. (Even if the group only has a single server, the server is still a member of a group.) It is the responsibility of the Domain Controller and the Host Controllers to ensure that all servers in a server group have a consistent configuration. They should all be configured with the same profile and they should have the same deployment content deployed.

The domain can have multiple server groups. The above diagram shows two server groups, "ServerGroupA" and "ServerGroupB". Different server groups can be configured with different profiles and deployments; for example in a domain with different tiers of servers providing different services. Different server groups can also run the same profile and have the same deployments; for example to support rolling application upgrade scenarios where a complete service outage is avoided by first upgrading the application on one server group and then upgrading a second server group.

An example server group definition is as follows:

```
<server-group name="main-server-group" profile="default">
    <socket-binding-group ref="standard-sockets"/>
    <deployments>
        <deployment name="foo.war_v1" runtime-name="foo.war" />
        <deployment name="bar.ear" runtime-name="bar.ear" />
    </deployments>
</server-group>
```

A server-group configuration includes the following required attributes:

- name -- the name of the server group
- profile -- the name of the profile the servers in the group should run

In addition, the following optional elements are available:

- socket-binding-group -- specifies the name of the default socket binding group to use on servers in the group. Can be overridden on a per-server basis in `host.xml`. If not provided in the `server-group` element, it must be provided for each server in `host.xml`.
- deployments -- the deployment content that should be deployed on the servers in the group.
- deployment-overlays -- the overlays and their associated deployments.
- system-properties -- system properties that should be set on all servers in the group
- jvm -- default jvm settings for all servers in the group. The Host Controller will merge these settings with any provided in `host.xml` to derive the settings to use to launch the server's JVM. See JVM settings for further details.

Server

Each "Server" in the above diagram represents an actual application server instance. The server runs in a separate JVM process from the Host Controller. The Host Controller is responsible for launching that process. (In a managed domain the end user cannot directly launch a server process from the command line.)

The Host Controller synthesizes the server's configuration by combining elements from the domain wide configuration (from `domain.xml`) and the host-specific configuration (from `host.xml`).

Deciding between running standalone servers or a managed domain

Which use cases are appropriate for managed domain and which are appropriate for standalone servers? A managed domain is all about coordinated multi-server management -- with it WildFly provides a central point through which users can manage multiple servers, with rich capabilities to keep those servers' configurations consistent and the ability to roll out configuration changes (including deployments) to the servers in a coordinated fashion.

It's important to understand that the choice between a managed domain and standalone servers is all about how your servers are managed, not what capabilities they have to service end user requests. This distinction is particularly important when it comes to high availability clusters. It's important to understand that HA functionality is orthogonal to running standalone servers or a managed domain. That is, a group of standalone servers can be configured to form an HA cluster. The domain and standalone modes determine how the servers are managed, not what capabilities they provide.

So, given all that:

- A single server installation gains nothing from running in a managed domain, so running a standalone server is a better choice.
- For multi-server production environments, the choice of running a managed domain versus standalone servers comes down to whether the user wants to use the centralized management capabilities a managed domain provides. Some enterprises have developed their own sophisticated multi-server management capabilities and are comfortable coordinating changes across a number of independent WildFly instances. For these enterprises, a multi-server architecture comprised of individual standalone servers is a good option.
- Running a standalone server is better suited for most development scenarios. Any individual server configuration that can be achieved in a managed domain can also be achieved in a standalone server, so even if the application being developed will eventually run in production on a managed domain installation, much (probably most) development can be done using a standalone server.
- Running a managed domain mode can be helpful in some advanced development scenarios; i.e. those involving interaction between multiple WildFly instances. Developers may find that setting up various servers as members of a domain is an efficient way to launch a multi-server cluster.

5.14.2 General configuration concepts

For both a managed domain or a standalone server, a number of common configuration concepts apply:

Extensions

An extension is a module that extends the core capabilities of the server. The WildFly core is very simple and lightweight; most of the capabilities people associate with an application server are provided via extensions. An extension is packaged as a module in the `modules` folder. The user indicates that they want a particular extension to be available by including an `<extension/>` element naming its module in the `domain.xml` or `standalone.xml` file.

```
<extensions>
    [...]
    <extension module="org.jboss.as.transactions"/>
    <extension module="org.jboss.as.webservices" />
    <extension module="org.jboss.as.weld" />
    [...]
    <extension module="org.wildfly.extension.undertow"/>
</extensions>
```

Profiles and Subsystems

The most significant part of the configuration in `domain.xml` and `standalone.xml` is the configuration of one (in `standalone.xml`) or more (in `domain.xml`) "profiles". A profile is a named set of subsystem configurations. A subsystem is an added set of capabilities added to the core server by an extension (see "Extensions" above). A subsystem provides servlet handling capabilities; a subsystem provides an EJB container; a subsystem provides JTA, etc. A profile is a named list of subsystems, along with the details of each subsystem's configuration. A profile with a large number of subsystems results in a server with a large set of capabilities. A profile with a small, focused set of subsystems will have fewer capabilities but a smaller footprint.

The content of an individual profile configuration looks largely the same in `domain.xml` and `standalone.xml`. The only difference is `standalone.xml` is only allowed to have a single profile element (the profile the server will run), while `domain.xml` can have many profiles, each of which can be mapped to one or more groups of servers.

The contents of individual subsystem configurations look exactly the same between `domain.xml` and `standalone.xml`.

Paths

A logical name for a filesystem path. The `domain.xml`, `host.xml` and `standalone.xml` configurations all include a section where paths can be declared. Other sections of the configuration can then reference those paths by their logical name, rather than having to include the full details of the path (which may vary on different machines). For example, the logging subsystem configuration includes a reference to the "`jboss.server.log.dir`" path that points to the server's "`log`" directory.

```
<file relative-to="jboss.server.log.dir" path="server.log"/>
```

WildFly automatically provides a number of standard paths without any need for the user to configure them in a configuration file:

- `jboss.home.dir` - the root directory of the WildFly distribution
- `user.home` - user's home directory
- `user.dir` - user's current working directory
- `java.home` - java installation directory
- `jboss.server.base.dir` - root directory for an individual server instance
- `jboss.server.config.dir` - directory the server will use for configuration file storage
- `jboss.server.data.dir` - directory the server will use for persistent data file storage
- `jboss.server.log.dir` - directory the server will use for log file storage
- `jboss.server.temp.dir` - directory the server will use for temporary file storage
- `jboss.controller.temp.dir` - directory the server will use for temporary file storage
- `jboss.domain.servers.dir` - directory under which a host controller will create the working area for individual server instances (managed domain mode only)

Users can add their own paths or override all except the first 5 of the above by adding a `<path/>` element to their configuration file.

```
<path name="example" path="example" relative-to="jboss.server.data.dir"/>
```

The attributes are:

- `name` -- the name of the path.
- `path` -- the actual filesystem path. Treated as an absolute path, unless the 'relative-to' attribute is specified, in which case the value is treated as relative to that path.
- `relative-to` -- (optional) the name of another previously named path, or of one of the standard paths provided by the system.

A `<path/>` element in a `domain.xml` need not include anything more than the `name` attribute; i.e. it need not include any information indicating what the actual filesystem path is:

```
<path name="x"/>
```

Such a configuration simply says, "There is a path named 'x' that other parts of the `domain.xml` configuration can reference. The actual filesystem location pointed to by 'x' is host-specific and will be specified in each machine's `host.xml` file." If this approach is used, there must be a path element in each machine's `host.xml` that specifies what the actual filesystem path is:

```
<path name="x" path="/var/x" />
```

A <path/> element in a standalone.xml must include the specification of the actual filesystem path.

Interfaces

A logical name for a network interface/IP address/host name to which sockets can be bound. The domain.xml, host.xml and standalone.xml configurations all include a section where interfaces can be declared. Other sections of the configuration can then reference those interfaces by their logical name, rather than having to include the full details of the interface (which may vary on different machines). An interface configuration includes the logical name of the interface as well as information specifying the criteria to use for resolving the actual physical address to use. See Interfaces and ports for further details.

An <interface/> element in a domain.xml need not include anything more than the name attribute; i.e. it need not include any information indicating what the actual IP address associated with the name is:

```
<interface name="internal"/>
```

Such a configuration simply says, "There is an interface named 'internal' that other parts of the domain.xml configuration can reference. The actual IP address pointed to by 'internal' is host-specific and will be specified in each machine's host.xml file." If this approach is used, there must be an interface element in each machine's host.xml that specifies the criteria for determining the IP address:

```
<interface name="internal">
   <nic name="eth1"/>
</interface>
```

An <interface/> element in a standalone.xml must include the criteria for determining the IP address.

Socket Bindings and Socket Binding Groups

A socket binding is a named configuration for a socket.

The domain.xml and standalone.xml configurations both include a section where named socket configurations can be declared. Other sections of the configuration can then reference those sockets by their logical name, rather than having to include the full details of the socket configuration (which may vary on different machines). See Interfaces and ports for full details.

System Properties

System property values can be set in a number of places in `domain.xml`, `host.xml` and `standalone.xml`. The values in `standalone.xml` are set as part of the server boot process. Values in `domain.xml` and `host.xml` are applied to servers when they are launched.

When a system property is configured in `domain.xml` or `host.xml`, the servers it ends up being applied to depends on where it is set. Setting a system property in a child element directly under the `domain.xml` root results in the property being set on all servers. Setting it in a `<system-property/>` element inside a `<server-group/>` element in domain.xml results in the property being set on all servers in the group. Setting it in a child element directly under the `host.xml` root results in the property being set on all servers controlled by that host's Host Controller. Finally, setting it in a `<system-property/>` element inside a `<server/>` element in `host.xml` result in the property being set on that server. The same property can be configured in multiple locations, with a value in a `<server/>` element taking precedence over a value specified directly under the `host.xml` root element, the value in a `host.xml` taking precedence over anything from `domain.xml`, and a value in a `<server-group/>` element taking precedence over a value specified directly under the `domain.xml` root element.

5.14.3 Management resources

When WildFly parses your configuration files at boot, or when you use one of the AS's Management Clients you are adding, removing or modifying *management resources* in the AS's internal management model. A WildFly management resource has the following characteristics:

Address

All WildFly management resources are organized in a tree. The path to the node in the tree for a particular resource is its *address*. Each segment in a resource's address is a key/value pair:

- The key is the resource's *type*, in the context of its parent. So, for example, the root resource for a standalone server has children of type `subsystem`, `interface`, `socket-binding`, etc. The resource for the subsystem that provides the AS's webserver capability has children of type `connector` and `virtual-server`. The resource for the subsystem that provides the AS's messaging server capability has, among others, children of type `jms-queue` and `jms-topic`.
- The value is the name of a particular resource of the given type, e.g `web` or `messaging` for subsystems or `http` or `https` for web subsystem connectors.

The full address for a resource is the ordered list of key/value pairs that lead from the root of the tree to the resource. Typical notation is to separate the elements in the address with a '/' and to separate the key and the value with an '=':

- `/subsystem=undertow/server=default-server/http-listener=default`
- `/subsystem=messaging/jms-queue=testQueue`
- `/interface=public`

When using the HTTP API, a '/' is used to separate the key and the value instead of an '=':

- `http://localhost:9990/management/subsystem/undertow/server/default-server/h`
- `http://localhost:9990/management/subsystem/messaging/jms-queue/testQueue`
- `http://localhost:9990/management/interface/public`

Operations

Querying or modifying the state of a resource is done via an operation. An operation has the following characteristics:

- A string name
- Zero or more named parameters. Each parameter has a string name, and a value of type `org.jboss.dmr.ModelNode` (or, when invoked via the CLI, the text representation of a `ModelNode`; when invoked via the HTTP API, the JSON representation of a `ModelNode`.) Parameters may be optional.
- A return value, which will be of type `org.jboss.dmr.ModelNode` (or, when invoked via the CLI, the text representation of a `ModelNode`; when invoked via the HTTP API, the JSON representation of a `ModelNode`.)

Every resource except the root resource will have an `add` operation and should have a `remove` operation ("should" because in WildFly 8 many do not). The parameters for the `add` operation vary depending on the resource. The `remove` operation has no parameters.

There are also a number of "global" operations that apply to all resources. See Global operations for full details.

The operations a resource supports can themselves be determined by invoking an operation: the `read-operation-names` operation. Once the name of an operation is known, details about its parameters and return value can be determined by invoking the `read-operation-description` operation. For example, to learn the names of the operations exposed by the root resource for a standalone server, and then learn the full details of one of them, via the CLI one would:

```
[standalone@localhost:9990 /] :read-operation-names
{
    "outcome" => "success",
    "result" => [
        "add-namespace",
        "add-schema-location",
        "delete-snapshot",
        "full-replace-deployment",
        "list-snapshots",
        "read-attribute",
        "read-children-names",
        "read-children-resources",
        "read-children-types",
        "read-config-as-xml",
        "read-operation-description",
        "read-operation-names",
        "read-resource",
        "read-resource-description",
        "reload",
        "remove-namespace",
        "remove-schema-location",
        "replace-deployment",
        "shutdown",
        "take-snapshot",
        "upload-deployment-bytes",
        "upload-deployment-stream",
        "upload-deployment-url",
        "validate-address",
        "write-attribute"
    ]
}
[standalone@localhost:9990 /] :read-operation-description(name=upload-deployment-url)
{
    "outcome" => "success",
    "result" => {
        "operation-name" => "upload-deployment-url",
        "description" => "Indicates that the deployment content available at the included URL
should be added to the deployment content repository. Note that this operation does not indicate
the content should be deployed into the runtime.",
        "request-properties" => {"url" => {
            "type" => STRING,
            "description" => "The URL at which the deployment content is available for upload to
the domain's or standalone server's deployment content repository.. Note that the URL must be
accessible from the target of the operation (i.e. the Domain Controller or standalone server).",
            "required" => true,
            "min-length" => 1,
            "nillable" => false
        }},
        "reply-properties" => {
            "type" => BYTES,
            "description" => "The hash of managed deployment content that has been uploaded to
the domain's or standalone server's deployment content repository.",
            "min-length" => 20,
            "max-length" => 20,
            "nillable" => false
        }
    }
}
```

See Descriptions below for more on how to learn about the operations a resource exposes.

Attributes

Management resources expose information about their state as attributes. Attributes have string name, and a value of type `org.jboss.dmr.ModelNode` (or: for the CLI, the text representation of a `ModelNode`; for HTTP API, the JSON representation of a `ModelNode`.)

Attributes can either be read-only or read-write. Reading and writing attribute values is done via the global `read-attribute` and `write-attribute` operations.

The `read-attribute` operation takes a single parameter "name" whose value is a the name of the attribute. For example, to read the "port" attribute of a socket-binding resource via the CLI:

```
[standalone@localhost:9990 /]
/socket-binding-group=standard-sockets/socket-binding=https:read-attribute(name=port)
{
    "outcome" => "success",
    "result" => 8443
}
```

If an attribute is writable, the `write-attribute` operation is used to mutate its state. The operation takes two parameters:

- `name` – the name of the attribute
- `value` – the value of the attribute

For example, to read the "port" attribute of a socket-binding resource via the CLI:

```
[standalone@localhost:9990 /]
/socket-binding-group=standard-sockets/socket-binding=https:write-attribute(name=port,value=8444)
=> "success"}
```

Attributes can have one of two possible *storage types*:

- **CONFIGURATION** – means the value of the attribute is stored in the persistent configuration; i.e. in the `domain.xml`, `host.xml` or `standalone.xml` file from which the resource's configuration was read.
- **RUNTIME** – the attribute value is only available from a running server; the value is not stored in the persistent configuration. A metric (e.g. number of requests serviced) is a typical example of a RUNTIME attribute.

The values of all of the attributes a resource exposes can be obtained via the `read-resource` operation, with the "include-runtime" parameter set to "true". For example, from the CLI:

```
[standalone@localhost:9990 /]
/subsystem=undertow/server=default-server/http-listener=default:read-resource(include-runtime=true
"outcome" => "success",
    "result" => {
        "allow-encoded-slash" => false,
        "allow-equals-in-cookie-value" => false,
        "always-set-keep-alive" => true,
        "buffer-pipelined-data" => true,
        "buffer-pool" => "default",
        "bytes-received" => 0L,
        "bytes-sent" => 0L,
        "certificate-forwarding" => false,
        "decode-url" => true,
        "disallowed-methods" => ["TRACE"],
        "enable-http2" => false,
        "enabled" => true,
        "error-count" => 0L,
        "max-buffered-request-size" => 16384,
        "max-connections" => undefined,
        "max-cookies" => 200,
        "max-header-size" => 1048576,
        "max-headers" => 200,
        "max-parameters" => 1000,
        "max-post-size" => 10485760L,
        "max-processing-time" => 0L,
        "no-request-timeout" => undefined,
        "processing-time" => 0L,
        "proxy-address-forwarding" => false,
        "read-timeout" => undefined,
        "receive-buffer" => undefined,
        "record-request-start-time" => false,
        "redirect-socket" => "https",
        "request-count" => 0L,
        "request-parse-timeout" => undefined,
        "resolve-peer-address" => false,
        "send-buffer" => undefined,
        "socket-binding" => "http",
        "tcp-backlog" => undefined,
        "tcp-keep-alive" => undefined,
        "url-charset" => "UTF-8",
        "worker" => "default",
        "write-timeout" => undefined
    }
}
```

Omit the "include-runtime" parameter (or set it to "false") to limit output to those attributes whose values are stored in the persistent configuration:

```
[standalone@localhost:9990 /]
/subsystem=undertow/server=default-server/http-listener=default:read-resource(include-runtime=fals
"outcome" => "success",
    "result" => {
        "allow-encoded-slash" => false,
        "allow-equals-in-cookie-value" => false,
        "always-set-keep-alive" => true,
        "buffer-pipelined-data" => true,
        "buffer-pool" => "default",
        "certificate-forwarding" => false,
        "decode-url" => true,
        "disallowed-methods" => ["TRACE"],
        "enable-http2" => false,
        "enabled" => true,
        "max-buffered-request-size" => 16384,
        "max-connections" => undefined,
        "max-cookies" => 200,
        "max-header-size" => 1048576,
        "max-headers" => 200,
        "max-parameters" => 1000,
        "max-post-size" => 10485760L,
        "no-request-timeout" => undefined,
        "proxy-address-forwarding" => false,
        "read-timeout" => undefined,
        "receive-buffer" => undefined,
        "record-request-start-time" => false,
        "redirect-socket" => "https",
        "request-parse-timeout" => undefined,
        "resolve-peer-address" => false,
        "send-buffer" => undefined,
        "socket-binding" => "http",
        "tcp-backlog" => undefined,
        "tcp-keep-alive" => undefined,
        "url-charset" => "UTF-8",
        "worker" => "default",
        "write-timeout" => undefined
    }
}
```

See Descriptions below for how to learn more about the attributes a particular resource exposes.

Children

Management resources may support child resources. The *types* of children a resource supports (e.g. `connector` for the web subsystem resource) can be obtained by querying the resource's description (see Descriptions below) or by invoking the `read-children-types` operation. Once you know the legal child types, you can query the names of all children of a given type by using the global `read-children-types` operation. The operation takes a single parameter "child-type" whose value is the type. For example, a resource representing a socket binding group has children. To find the type of those children and the names of resources of that type via the CLI one could:

```
[standalone@localhost:9990 /] /socket-binding-group=standard-sockets:read-children-types
{
    "outcome" => "success",
    "result" => ["socket-binding"]
}
[standalone@localhost:9990 /]
/socket-binding-group=standard-sockets:read-children-names(child-type=socket-binding)
{
    "outcome" => "success",
    "result" => [
        "http",
        "https",
        "jmx-connector-registry",
        "jmx-connector-server",
        "jndi",
        "osgi-http",
        "remoting",
        "txn-recovery-environment",
        "txn-status-manager"
    ]
}
```

Descriptions

All resources expose metadata that describes their attributes, operations and child types. This metadata is itself obtained by invoking one or more of the global operations each resource supports. We showed examples of the `read-operation-names`, `read-operation-description`, `read-children-types` and `read-children-names` operations above.

The `read-resource-description` operation can be used to find the details of the attributes and child types associated with a resource. For example, using the CLI:

```
[standalone@localhost:9990 /] /socket-binding-group=standard-sockets:read-resource-description
{
    "outcome" => "success",
    "result" => {
        "description" => "Contains a list of socket configurations.",
        "head-comment-allowed" => true,
        "tail-comment-allowed" => false,
        "attributes" => {
            "name" => {
                "type" => STRING,
                "description" => "The name of the socket binding group.",
                "required" => true,
                "head-comment-allowed" => false,
                "tail-comment-allowed" => false,
                "access-type" => "read-only",
                "storage" => "configuration"
            },
            "default-interface" => {
                "type" => STRING,
                "description" => "Name of an interface that should be used as the interface for
any sockets that do not explicitly declare one.",
                "required" => true,
                "head-comment-allowed" => false,
                "tail-comment-allowed" => false,
                "access-type" => "read-write",
                "storage" => "configuration"
            },
            "port-offset" => {
                "type" => INT,
                "description" => "Increment to apply to the base port values defined in the
socket bindings to derive the runtime values to use on this server.",
                "required" => false,
                "head-comment-allowed" => true,
                "tail-comment-allowed" => false,
                "access-type" => "read-write",
                "storage" => "configuration"
            }
        },
        "operations" => {},
        "children" => {"socket-binding" => {
            "description" => "The individual socket configurtions.",
            "min-occurs" => 0,
            "model-description" => undefined
        }}
    }
}
```

Note the `"operations" => }}` in the output above. If the command had included the `{{operations` parameter (i.e. `/socket-binding-group=standard-sockets:read-resource-description(operations=true`) the output would have included the description of each operation supported by the resource.

See the Global operations section for details on other parameters supported by the `read-resource-description` operation and all the other globally available operations.

Comparison to JMX MBeans

WildFly management resources are conceptually quite similar to Open MBeans. They have the following primary differences:

- WildFly management resources are organized in a tree structure. The order of the key value pairs in a resource's address is significant, as it defines the resource's position in the tree. The order of the key properties in a JMX `ObjectName` is not significant.
- In an Open MBean attribute values, operation parameter values and operation return values must either be one of the simple JDK types (String, Boolean, Integer, etc) or implement either the `javax.management.openmbean.CompositeData` interface or the `javax.management.openmbean.TabularData` interface. WildFly management resource attribute values, operation parameter values and operation return values are all of type `org.jboss.dmr.ModelNode`.

Basic structure of the management resource trees

As noted above, management resources are organized in a tree structure. The structure of the tree depends on whether you are running a standalone server or a managed domain.

Standalone server

The structure of the managed resource tree is quite close to the structure of the `standalone.xml` configuration file.

- The root resource
  - `extension` – extensions installed in the server
  - `path` – paths available on the server
  - `system-property` – system properties set as part of the configuration (i.e. not on the command line)
  - `core-service=management` – the server's core management services
  - `core-service=service-container` – resource for the JBoss MSC `ServiceContainer` that's at the heart of the AS
  - `subsystem` – the subsystems installed on the server. The bulk of the management model will be children of type `subsystem`
  - `interface` – interface configurations
  - `socket-binding-group` – the central resource for the server's socket bindings
    - `socket-binding` – individual socket binding configurations
  - `deployment` – available deployments on the server

Managed domain

In a managed domain, the structure of the managed resource tree spans the entire domain, covering both the domain wide configuration (e.g. what's in `domain.xml`, the host specific configuration for each host (e.g. what's in `host.xml`, and the resources exposed by each running application server. The Host Controller processes in a managed domain provide access to all or part of the overall resource tree. How much is available depends on whether the management client is interacting with the Host Controller that is acting as the master Domain Controller. If the Host Controller is the master Domain Controller, then the section of the tree for each host is available. If the Host Controller is a slave to a remote Domain Controller, then only the portion of the tree associated with that host is available.

- The root resource for the entire domain. The persistent configuration associated with this resource and its children, except for those of type `host`, is persisted in the `domain.xml` file on the Domain Controller.

- `extension` – extensions available in the domain
- `path` – paths available on across the domain
- `system-property` – system properties set as part of the configuration (i.e. not on the command line) and available across the domain
- `profile` – sets of subsystem configurations that can be assigned to server groups
  - `subsystem` – configuration of subsystems that are part of the profile
- `interface` – interface configurations
- `socket-binding-group` – sets of socket bindings configurations that can be applied to server groups
  - `socket-binding` – individual socket binding configurations
- `deployment` – deployments available for assignment to server groups
- deployment-overlay -- deployment-overlays content available to overlay deployments in server groups
- `server-group` – server group configurations
- host – the individual Host Controllers. Each child of this type represents the root resource for a particular host. The persistent configuration associated with one of these resources or its children is persisted in the host's `host.xml` file.
  - `path` – paths available on each server on the host
  - `system-property` – system properties to set on each server on the host
  - `core-service=management` – the Host Controller's core management services
  - `interface` – interface configurations that apply to the Host Controller or servers on the host
  - `jvm` – JVM configurations that can be applied when launching servers
  - `server-config` – configuration describing how the Host Controller should launch a server; what server group configuration to use, and any server-specific overrides of items specified in other resources
  - `server` – the root resource for a running server. Resources from here and below are not directly persisted; the domain-wide and host level resources contain the persistent configuration that drives a server
    - `extension` – extensions installed in the server
    - `path` – paths available on the server
    - `system-property` – system properties set as part of the configuration (i.e. not on the command line)
    - `core-service=management` – the server's core management services
    - `core-service=service-container` – resource for the JBoss MSC `ServiceContainer` that's at the heart of the AS
    - `subsystem` – the subsystems installed on the server. The bulk of the management model will be children of type `subsystem`
    - `interface` – interface configurations
    - `socket-binding-group` – the central resource for the server's socket bindings
      - `socket-binding` – individual socket binding configurations
    - `deployment` – available deployments on the server
    - deployment-overlay -- available overlays on the server

5.14.4 General configuration concepts

For both a managed domain or a standalone server, a number of common configuration concepts apply:

Extensions

An extension is a module that extends the core capabilities of the server. The WildFly core is very simple and lightweight; most of the capabilities people associate with an application server are provided via extensions. An extension is packaged as a module in the `modules` folder. The user indicates that they want a particular extension to be available by including an `<extension/>` element naming its module in the `domain.xml` or `standalone.xml` file.

```
<extensions>
    [...]
    <extension module="org.jboss.as.transactions"/>
    <extension module="org.jboss.as.webservices" />
    <extension module="org.jboss.as.weld" />
    [...]
    <extension module="org.wildfly.extension.undertow"/>
</extensions>
```

Profiles and Subsystems

The most significant part of the configuration in `domain.xml` and `standalone.xml` is the configuration of one (in `standalone.xml`) or more (in `domain.xml`) "profiles". A profile is a named set of subsystem configurations. A subsystem is an added set of capabilities added to the core server by an extension (see "Extensions" above). A subsystem provides servlet handling capabilities; a subsystem provides an EJB container; a subsystem provides JTA, etc. A profile is a named list of subsystems, along with the details of each subsystem's configuration. A profile with a large number of subsystems results in a server with a large set of capabilities. A profile with a small, focused set of subsystems will have fewer capabilities but a smaller footprint.

The content of an individual profile configuration looks largely the same in `domain.xml` and `standalone.xml`. The only difference is `standalone.xml` is only allowed to have a single profile element (the profile the server will run), while `domain.xml` can have many profiles, each of which can be mapped to one or more groups of servers.

The contents of individual subsystem configurations look exactly the same between `domain.xml` and `standalone.xml`.

Paths

A logical name for a filesystem path. The `domain.xml`, `host.xml` and `standalone.xml` configurations all include a section where paths can be declared. Other sections of the configuration can then reference those paths by their logical name, rather than having to include the full details of the path (which may vary on different machines). For example, the logging subsystem configuration includes a reference to the "`jboss.server.log.dir`" path that points to the server's "`log`" directory.

```
<file relative-to="jboss.server.log.dir" path="server.log"/>
```

WildFly automatically provides a number of standard paths without any need for the user to configure them in a configuration file:

- `jboss.home`.dir - the root directory of the WildFly distribution
- `user.home` - user's home directory
- `user.dir` - user's current working directory
- `java.home` - java installation directory
- `jboss.server.base.dir` - root directory for an individual server instance
- `jboss.server.config.dir` - directory the server will use for configuration file storage
- `jboss.server.data.dir` - directory the server will use for persistent data file storage
- `jboss.server.log.dir` - directory the server will use for log file storage
- `jboss.server.temp.dir` - directory the server will use for temporary file storage
- `jboss.controller.temp.dir` - directory the server will use for temporary file storage
- `jboss.domain.servers.dir` - directory under which a host controller will create the working area for individual server instances (managed domain mode only)

Users can add their own paths or override all except the first 5 of the above by adding a `<path/>` element to their configuration file.

```
<path name="example" path="example" relative-to="jboss.server.data.dir"/>
```

The attributes are:

- `name` -- the name of the path.
- `path` -- the actual filesystem path. Treated as an absolute path, unless the 'relative-to' attribute is specified, in which case the value is treated as relative to that path.
- `relative-to` -- (optional) the name of another previously named path, or of one of the standard paths provided by the system.

A `<path/>` element in a `domain.xml` need not include anything more than the `name` attribute; i.e. it need not include any information indicating what the actual filesystem path is:

```
<path name="x"/>
```

Such a configuration simply says, "There is a path named 'x' that other parts of the `domain.xml` configuration can reference. The actual filesystem location pointed to by 'x' is host-specific and will be specified in each machine's `host.xml` file." If this approach is used, there must be a path element in each machine's `host.xml` that specifies what the actual filesystem path is:

```
<path name="x" path="/var/x" />
```

A `<path/>` element in a `standalone.xml` must include the specification of the actual filesystem path.

Interfaces

A logical name for a network interface/IP address/host name to which sockets can be bound. The `domain.xml`, `host.xml` and `standalone.xml` configurations all include a section where interfaces can be declared. Other sections of the configuration can then reference those interfaces by their logical name, rather than having to include the full details of the interface (which may vary on different machines). An interface configuration includes the logical name of the interface as well as information specifying the criteria to use for resolving the actual physical address to use. See Interfaces and ports for further details.

An `<interface/>` element in a `domain.xml` need not include anything more than the `name` attribute; i.e. it need not include any information indicating what the actual IP address associated with the name is:

```
<interface name="internal"/>
```

Such a configuration simply says, "There is an interface named 'internal' that other parts of the domain.xml configuration can reference. The actual IP address pointed to by 'internal' is host-specific and will be specified in each machine's host.xml file." If this approach is used, there must be an interface element in each machine's `host.xml` that specifies the criteria for determining the IP address:

```
<interface name="internal">
   <nic name="eth1"/>
</interface>
```

An `<interface/>` element in a `standalone.xml` must include the criteria for determining the IP address.

Socket Bindings and Socket Binding Groups

A socket binding is a named configuration for a socket.

The `domain.xml` and `standalone.xml` configurations both include a section where named socket configurations can be declared. Other sections of the configuration can then reference those sockets by their logical name, rather than having to include the full details of the socket configuration (which may vary on different machines). See Interfaces and ports for full details.

System Properties

System property values can be set in a number of places in `domain.xml`, `host.xml` and `standalone.xml`. The values in `standalone.xml` are set as part of the server boot process. Values in `domain.xml` and `host.xml` are applied to servers when they are launched.

When a system property is configured in `domain.xml` or `host.xml`, the servers it ends up being applied to depends on where it is set. Setting a system property in a child element directly under the `domain.xml` root results in the property being set on all servers. Setting it in a `<system-property/>` element inside a `<server-group/>` element in domain.xml results in the property being set on all servers in the group. Setting it in a child element directly under the `host.xml` root results in the property being set on all servers controlled by that host's Host Controller. Finally, setting it in a `<system-property/>` element inside a `<server/>` element in `host.xml` result in the property being set on that server. The same property can be configured in multiple locations, with a value in a `<server/>` element taking precedence over a value specified directly under the `host.xml` root element, the value in a `host.xml` taking precedence over anything from `domain.xml`, and a value in a `<server-group/>` element taking precedence over a value specified directly under the `domain.xml` root element.

5.14.5 Management resources

When WildFly parses your configuration files at boot, or when you use one of the AS's Management Clients you are adding, removing or modifying *management resources* in the AS's internal management model. A WildFly management resource has the following characteristics:

Address

All WildFly management resources are organized in a tree. The path to the node in the tree for a particular resource is its *address*. Each segment in a resource's address is a key/value pair:

- The key is the resource's *type*, in the context of its parent. So, for example, the root resource for a standalone server has children of type `subsystem`, `interface`, `socket-binding`, etc. The resource for the subsystem that provides the AS's webserver capability has children of type `connector` and `virtual-server`. The resource for the subsystem that provides the AS's messaging server capability has, among others, children of type `jms-queue` and `jms-topic`.
- The value is the name of a particular resource of the given type, e.g `web` or `messaging` for subsystems or `http` or `https` for web subsystem connectors.

The full address for a resource is the ordered list of key/value pairs that lead from the root of the tree to the resource. Typical notation is to separate the elements in the address with a '/' and to separate the key and the value with an '=':

- `/subsystem=undertow/server=default-server/http-listener=default`
- `/subsystem=messaging/jms-queue=testQueue`
- `/interface=public`

When using the HTTP API, a '/' is used to separate the key and the value instead of an '=':

- `http://localhost:9990/management/subsystem/undertow/server/default-server/h`
- `http://localhost:9990/management/subsystem/messaging/jms-queue/testQueue`
- `http://localhost:9990/management/interface/public`

Operations

Querying or modifying the state of a resource is done via an operation. An operation has the following characteristics:

- A string name
- Zero or more named parameters. Each parameter has a string name, and a value of type `org.jboss.dmr.ModelNode` (or, when invoked via the CLI, the text representation of a `ModelNode`; when invoked via the HTTP API, the JSON representation of a `ModelNode`.) Parameters may be optional.
- A return value, which will be of type `org.jboss.dmr.ModelNode` (or, when invoked via the CLI, the text representation of a `ModelNode`; when invoked via the HTTP API, the JSON representation of a `ModelNode`.)

Every resource except the root resource will have an `add` operation and should have a `remove` operation ("should" because in WildFly 8 many do not). The parameters for the `add` operation vary depending on the resource. The `remove` operation has no parameters.

There are also a number of "global" operations that apply to all resources. See Global operations for full details.

The operations a resource supports can themselves be determined by invoking an operation: the `read-operation-names` operation. Once the name of an operation is known, details about its parameters and return value can be determined by invoking the `read-operation-description` operation. For example, to learn the names of the operations exposed by the root resource for a standalone server, and then learn the full details of one of them, via the CLI one would:

```
[standalone@localhost:9990 /] :read-operation-names
{
    "outcome" => "success",
    "result" => [
        "add-namespace",
        "add-schema-location",
        "delete-snapshot",
        "full-replace-deployment",
        "list-snapshots",
        "read-attribute",
        "read-children-names",
        "read-children-resources",
        "read-children-types",
        "read-config-as-xml",
        "read-operation-description",
        "read-operation-names",
        "read-resource",
        "read-resource-description",
        "reload",
        "remove-namespace",
        "remove-schema-location",
        "replace-deployment",
        "shutdown",
        "take-snapshot",
        "upload-deployment-bytes",
        "upload-deployment-stream",
        "upload-deployment-url",
        "validate-address",
        "write-attribute"
    ]
}
[standalone@localhost:9990 /] :read-operation-description(name=upload-deployment-url)
{
    "outcome" => "success",
    "result" => {
        "operation-name" => "upload-deployment-url",
        "description" => "Indicates that the deployment content available at the included URL
should be added to the deployment content repository. Note that this operation does not indicate
the content should be deployed into the runtime.",
        "request-properties" => {"url" => {
            "type" => STRING,
            "description" => "The URL at which the deployment content is available for upload to
the domain's or standalone server's deployment content repository.. Note that the URL must be
accessible from the target of the operation (i.e. the Domain Controller or standalone server).",
            "required" => true,
            "min-length" => 1,
            "nillable" => false
        }},
        "reply-properties" => {
            "type" => BYTES,
            "description" => "The hash of managed deployment content that has been uploaded to
the domain's or standalone server's deployment content repository.",
            "min-length" => 20,
            "max-length" => 20,
            "nillable" => false
        }
    }
}
```

See Descriptions below for more on how to learn about the operations a resource exposes.

Attributes

Management resources expose information about their state as attributes. Attributes have string name, and a value of type `org.jboss.dmr.ModelNode` (or: for the CLI, the text representation of a `ModelNode`; for HTTP API, the JSON representation of a `ModelNode`.)

Attributes can either be read-only or read-write. Reading and writing attribute values is done via the global `read-attribute` and `write-attribute` operations.

The `read-attribute` operation takes a single parameter "name" whose value is a the name of the attribute. For example, to read the "port" attribute of a socket-binding resource via the CLI:

```
[standalone@localhost:9990 /]
/socket-binding-group=standard-sockets/socket-binding=https:read-attribute(name=port)
{
    "outcome" => "success",
    "result" => 8443
}
```

If an attribute is writable, the `write-attribute` operation is used to mutate its state. The operation takes two parameters:

- `name` – the name of the attribute
- `value` – the value of the attribute

For example, to read the "port" attribute of a socket-binding resource via the CLI:

```
[standalone@localhost:9990 /]
/socket-binding-group=standard-sockets/socket-binding=https:write-attribute(name=port,value=8444)
=> "success"}
```

Attributes can have one of two possible *storage types*:

- **CONFIGURATION** – means the value of the attribute is stored in the persistent configuration; i.e. in the `domain.xml`, `host.xml` or `standalone.xml` file from which the resource's configuration was read.
- **RUNTIME** – the attribute value is only available from a running server; the value is not stored in the persistent configuration. A metric (e.g. number of requests serviced) is a typical example of a RUNTIME attribute.

The values of all of the attributes a resource exposes can be obtained via the `read-resource` operation, with the "include-runtime" parameter set to "true". For example, from the CLI:

```
[standalone@localhost:9990 /]
/subsystem=undertow/server=default-server/http-listener=default:read-resource(include-runtime=true
"outcome" => "success",
    "result" => {
        "allow-encoded-slash" => false,
        "allow-equals-in-cookie-value" => false,
        "always-set-keep-alive" => true,
        "buffer-pipelined-data" => true,
        "buffer-pool" => "default",
        "bytes-received" => 0L,
        "bytes-sent" => 0L,
        "certificate-forwarding" => false,
        "decode-url" => true,
        "disallowed-methods" => ["TRACE"],
        "enable-http2" => false,
        "enabled" => true,
        "error-count" => 0L,
        "max-buffered-request-size" => 16384,
        "max-connections" => undefined,
        "max-cookies" => 200,
        "max-header-size" => 1048576,
        "max-headers" => 200,
        "max-parameters" => 1000,
        "max-post-size" => 10485760L,
        "max-processing-time" => 0L,
        "no-request-timeout" => undefined,
        "processing-time" => 0L,
        "proxy-address-forwarding" => false,
        "read-timeout" => undefined,
        "receive-buffer" => undefined,
        "record-request-start-time" => false,
        "redirect-socket" => "https",
        "request-count" => 0L,
        "request-parse-timeout" => undefined,
        "resolve-peer-address" => false,
        "send-buffer" => undefined,
        "socket-binding" => "http",
        "tcp-backlog" => undefined,
        "tcp-keep-alive" => undefined,
        "url-charset" => "UTF-8",
        "worker" => "default",
        "write-timeout" => undefined
    }
}
```

Omit the "include-runtime" parameter (or set it to "false") to limit output to those attributes whose values are stored in the persistent configuration:

```
[standalone@localhost:9990 /]
/subsystem=undertow/server=default-server/http-listener=default:read-resource(include-runtime=fals
"outcome" => "success",
    "result" => {
        "allow-encoded-slash" => false,
        "allow-equals-in-cookie-value" => false,
        "always-set-keep-alive" => true,
        "buffer-pipelined-data" => true,
        "buffer-pool" => "default",
        "certificate-forwarding" => false,
        "decode-url" => true,
        "disallowed-methods" => ["TRACE"],
        "enable-http2" => false,
        "enabled" => true,
        "max-buffered-request-size" => 16384,
        "max-connections" => undefined,
        "max-cookies" => 200,
        "max-header-size" => 1048576,
        "max-headers" => 200,
        "max-parameters" => 1000,
        "max-post-size" => 10485760L,
        "no-request-timeout" => undefined,
        "proxy-address-forwarding" => false,
        "read-timeout" => undefined,
        "receive-buffer" => undefined,
        "record-request-start-time" => false,
        "redirect-socket" => "https",
        "request-parse-timeout" => undefined,
        "resolve-peer-address" => false,
        "send-buffer" => undefined,
        "socket-binding" => "http",
        "tcp-backlog" => undefined,
        "tcp-keep-alive" => undefined,
        "url-charset" => "UTF-8",
        "worker" => "default",
        "write-timeout" => undefined
    }
}
```

See Descriptions below for how to learn more about the attributes a particular resource exposes.

Children

Management resources may support child resources. The *types* of children a resource supports (e.g. `connector` for the web subsystem resource) can be obtained by querying the resource's description (see Descriptions below) or by invoking the `read-children-types` operation. Once you know the legal child types, you can query the names of all children of a given type by using the global `read-children-types` operation. The operation takes a single parameter "child-type" whose value is the type. For example, a resource representing a socket binding group has children. To find the type of those children and the names of resources of that type via the CLI one could:

```
[standalone@localhost:9990 /] /socket-binding-group=standard-sockets:read-children-types
{
    "outcome" => "success",
    "result" => ["socket-binding"]
}
[standalone@localhost:9990 /]
/socket-binding-group=standard-sockets:read-children-names(child-type=socket-binding)
{
    "outcome" => "success",
    "result" => [
        "http",
        "https",
        "jmx-connector-registry",
        "jmx-connector-server",
        "jndi",
        "osgi-http",
        "remoting",
        "txn-recovery-environment",
        "txn-status-manager"
    ]
}
```

Descriptions

All resources expose metadata that describes their attributes, operations and child types. This metadata is itself obtained by invoking one or more of the global operations each resource supports. We showed examples of the `read-operation-names`, `read-operation-description`, `read-children-types` and `read-children-names` operations above.

The `read-resource-description` operation can be used to find the details of the attributes and child types associated with a resource. For example, using the CLI:

```
[standalone@localhost:9990 /] /socket-binding-group=standard-sockets:read-resource-description
{
    "outcome" => "success",
    "result" => {
        "description" => "Contains a list of socket configurations.",
        "head-comment-allowed" => true,
        "tail-comment-allowed" => false,
        "attributes" => {
            "name" => {
                "type" => STRING,
                "description" => "The name of the socket binding group.",
                "required" => true,
                "head-comment-allowed" => false,
                "tail-comment-allowed" => false,
                "access-type" => "read-only",
                "storage" => "configuration"
            },
            "default-interface" => {
                "type" => STRING,
                "description" => "Name of an interface that should be used as the interface for
any sockets that do not explicitly declare one.",
                "required" => true,
                "head-comment-allowed" => false,
                "tail-comment-allowed" => false,
                "access-type" => "read-write",
                "storage" => "configuration"
            },
            "port-offset" => {
                "type" => INT,
                "description" => "Increment to apply to the base port values defined in the
socket bindings to derive the runtime values to use on this server.",
                "required" => false,
                "head-comment-allowed" => true,
                "tail-comment-allowed" => false,
                "access-type" => "read-write",
                "storage" => "configuration"
            }
        },
        "operations" => {},
        "children" => {"socket-binding" => {
            "description" => "The individual socket configurtions.",
            "min-occurs" => 0,
            "model-description" => undefined
        }}
    }
}
```

Note the `"operations" => }}` in the output above. If the command had included the `{{operations` parameter (i.e. `/socket-binding-group=standard-sockets:read-resource-description(operations=true`) the output would have included the description of each operation supported by the resource.

See the Global operations section for details on other parameters supported by the `read-resource-description` operation and all the other globally available operations.

Comparison to JMX MBeans

WildFly management resources are conceptually quite similar to Open MBeans. They have the following primary differences:

- WildFly management resources are organized in a tree structure. The order of the key value pairs in a resource's address is significant, as it defines the resource's position in the tree. The order of the key properties in a JMX `ObjectName` is not significant.
- In an Open MBean attribute values, operation parameter values and operation return values must either be one of the simple JDK types (String, Boolean, Integer, etc) or implement either the `javax.management.openmbean.CompositeData` interface or the `javax.management.openmbean.TabularData` interface. WildFly management resource attribute values, operation parameter values and operation return values are all of type `org.jboss.dmr.ModelNode`.

Basic structure of the management resource trees

As noted above, management resources are organized in a tree structure. The structure of the tree depends on whether you are running a standalone server or a managed domain.

Standalone server

The structure of the managed resource tree is quite close to the structure of the `standalone.xml` configuration file.

- The root resource
  - `extension` – extensions installed in the server
  - `path` – paths available on the server
  - `system-property` – system properties set as part of the configuration (i.e. not on the command line)
  - `core-service=management` – the server's core management services
  - `core-service=service-container` – resource for the JBoss MSC `ServiceContainer` that's at the heart of the AS
  - `subsystem` – the subsystems installed on the server. The bulk of the management model will be children of type `subsystem`
  - `interface` – interface configurations
  - `socket-binding-group` – the central resource for the server's socket bindings
    - `socket-binding` – individual socket binding configurations
  - `deployment` – available deployments on the server

Managed domain

In a managed domain, the structure of the managed resource tree spans the entire domain, covering both the domain wide configuration (e.g. what's in `domain.xml`, the host specific configuration for each host (e.g. what's in `host.xml`, and the resources exposed by each running application server. The Host Controller processes in a managed domain provide access to all or part of the overall resource tree. How much is available depends on whether the management client is interacting with the Host Controller that is acting as the master Domain Controller. If the Host Controller is the master Domain Controller, then the section of the tree for each host is available. If the Host Controller is a slave to a remote Domain Controller, then only the portion of the tree associated with that host is available.

- The root resource for the entire domain. The persistent configuration associated with this resource and its children, except for those of type `host`, is persisted in the `domain.xml` file on the Domain Controller.

- `extension` – extensions available in the domain
- `path` – paths available on across the domain
- `system-property` – system properties set as part of the configuration (i.e. not on the command line) and available across the domain
- `profile` – sets of subsystem configurations that can be assigned to server groups
  - `subsystem` – configuration of subsystems that are part of the profile
- `interface` – interface configurations
- `socket-binding-group` – sets of socket bindings configurations that can be applied to server groups
  - `socket-binding` – individual socket binding configurations
- `deployment` – deployments available for assignment to server groups
- deployment-overlay -- deployment-overlays content available to overlay deployments in server groups
- `server-group` – server group configurations
- host – the individual Host Controllers. Each child of this type represents the root resource for a particular host. The persistent configuration associated with one of these resources or its children is persisted in the host's `host.xml` file.
  - `path` – paths available on each server on the host
  - `system-property` – system properties to set on each server on the host
  - `core-service=management` – the Host Controller's core management services
  - `interface` – interface configurations that apply to the Host Controller or servers on the host
  - `jvm` – JVM configurations that can be applied when launching servers
  - `server-config` – configuration describing how the Host Controller should launch a server; what server group configuration to use, and any server-specific overrides of items specified in other resources
  - `server` – the root resource for a running server. Resources from here and below are not directly persisted; the domain-wide and host level resources contain the persistent configuration that drives a server
    - `extension` – extensions installed in the server
    - `path` – paths available on the server
    - `system-property` – system properties set as part of the configuration (i.e. not on the command line)
    - `core-service=management` – the server's core management services
    - `core-service=service-container` – resource for the JBoss MSC `ServiceContainer` that's at the heart of the AS
    - `subsystem` – the subsystems installed on the server. The bulk of the management model will be children of type `subsystem`
    - `interface` – interface configurations
    - `socket-binding-group` – the central resource for the server's socket bindings
      - `socket-binding` – individual socket binding configurations
    - `deployment` – available deployments on the server
    - deployment-overlay -- available overlays on the server

5.14.6 Operating modes

WildFly can be booted in two different modes. A *managed domain* allows you to run and manage a multi-server topology. Alternatively, you can run a *standalone server* instance.

Standalone Server

For many use cases, the centralized management capability available via a managed domain is not necessary. For these use cases, a WildFly instance can be run as a "standalone server". A standalone server instance is an independent process, much like an JBoss Application Server 3, 4, 5, or 6 instance is. Standalone instances can be launched via the `standalone.sh` or `standalone.bat` launch scripts.

If more than one standalone instance is launched and multi-server management is desired, it is the user's responsibility to coordinate management across the servers. For example, to deploy an application across all of the standalone servers, the user would need to individually deploy the application on each server.

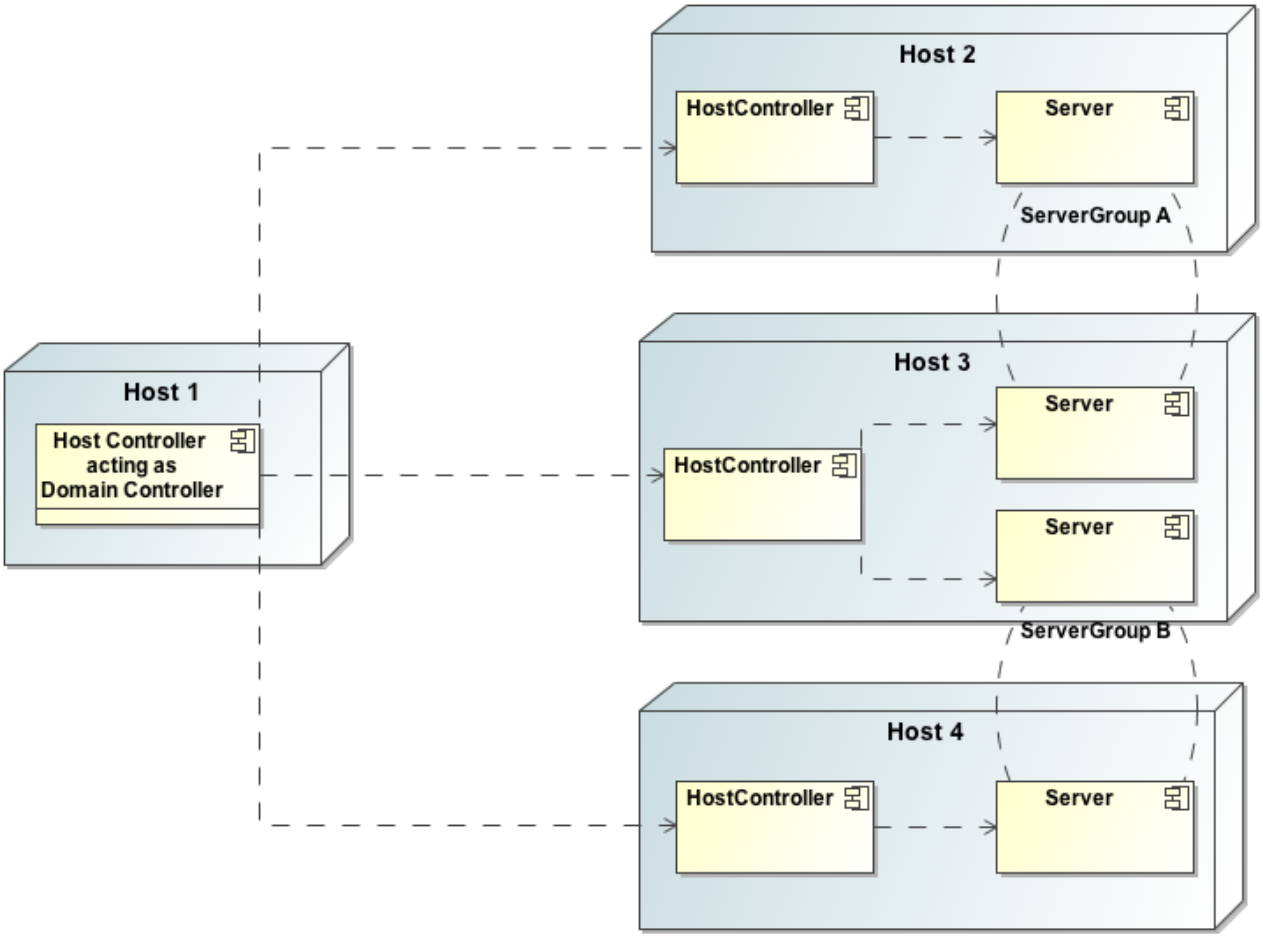
It is perfectly possible to launch multiple standalone server instances and have them form an HA cluster, just like it was possible with JBoss Application Server 3, 4, 5 and 6.

Managed Domain

One of the primary new features of WildFly is the ability to manage multiple WildFly instances from a single control point. A collection of such servers is referred to as the members of a "domain" with a single Domain Controller process acting as the central management control point. All of the WildFly instances in the domain share a common management policy, with the Domain Controller acting to ensure that each server is configured according to that policy. Domains can span multiple physical (or virtual) machines, with all WildFly instances on a given host under the control of a special Host Controller process. One Host Controller instance is configured to act as the central Domain Controller. The Host Controller on each host interacts with the Domain Controller to control the lifecycle of the application server instances running on its host and to assist the Domain Controller in managing them.

When you launch a WildFly managed domain on a host (via the `domain.sh` or `domain.bat` launch scripts) your intent is to launch a Host Controller and usually at least one WildFly instance. On one of the hosts the Host Controller should be configured to act as the Domain Controller. See Domain Setup for details.

The following is an example managed domain topology:



Host

Each "Host" box in the above diagram represents a physical or virtual host. A physical host can contain zero, one or more server instances.

Host Controller

When the domain.sh or domain.bat script is run on a host, a process known as a Host Controller is launched. The Host Controller is solely concerned with server management; it does not itself handle application server workloads. The Host Controller is responsible for starting and stopping the individual application server processes that run on its host, and interacts with the Domain Controller to help manage them.

Each Host Controller by default reads its configuration from the `domain/configuration/host.xml` file located in the unzipped WildFly installation on its host's filesystem. The `host.xml` file contains configuration information that is specific to the particular host. Primarily:

- the listing of the names of the actual WildFly instances that are meant to run off of this installation.
- configuration of how the Host Controller is to contact the Domain Controller to register itself and access the domain configuration. This may either be configuration of how to find and contact a remote Domain Controller, or a configuration telling the Host Controller to itself act as the Domain Controller.
- configuration of items that are specific to the local physical installation. For example, named interface definitions declared in `domain.xml` (see below) can be mapped to an actual machine-specific IP address in `host.xml`. Abstract path names in `domain.xml` can be mapped to actual filesystem paths in `host.xml`.

Domain Controller

One Host Controller instance is configured to act as the central management point for the entire domain, i.e. to be the Domain Controller. The primary responsibility of the Domain Controller is to maintain the domain's central management policy, to ensure all Host Controllers are aware of its current contents, and to assist the Host Controllers in ensuring any running application server instances are configured in accordance with this policy. This central management policy is stored by default in the `domain/configuration/domain.xml` file in the unzipped WildFly installation on Domain Controller's host's filesystem.

A `domain.xml` file must be located in the `domain/configuration` directory of an installation that's meant to run the Domain Controller. It does not need to be present in installations that are not meant to run a Domain Controller; i.e. those whose Host Controller is configured to contact a remote Domain Controller. The presence of a `domain.xml` file on such a server does no harm.

The `domain.xml` file includes, among other things, the configuration of the various "profiles" that WildFly instances in the domain can be configured to run. A profile configuration includes the detailed configuration of the various subsystems that comprise that profile (e.g. an embedded JBoss Web instance is a subsystem; a JBoss TS transaction manager is a subsystem, etc). The domain configuration also includes the definition of groups of sockets that those subsystems may open. The domain configuration also includes the definition of "server groups":

Server Group

A server group is set of server instances that will be managed and configured as one. In a managed domain each application server instance is a member of a server group. (Even if the group only has a single server, the server is still a member of a group.) It is the responsibility of the Domain Controller and the Host Controllers to ensure that all servers in a server group have a consistent configuration. They should all be configured with the same profile and they should have the same deployment content deployed.

The domain can have multiple server groups. The above diagram shows two server groups, "ServerGroupA" and "ServerGroupB". Different server groups can be configured with different profiles and deployments; for example in a domain with different tiers of servers providing different services. Different server groups can also run the same profile and have the same deployments; for example to support rolling application upgrade scenarios where a complete service outage is avoided by first upgrading the application on one server group and then upgrading a second server group.

An example server group definition is as follows:

```
<server-group name="main-server-group" profile="default">
    <socket-binding-group ref="standard-sockets"/>
    <deployments>
        <deployment name="foo.war_v1" runtime-name="foo.war" />
        <deployment name="bar.ear" runtime-name="bar.ear" />
    </deployments>
</server-group>
```

A server-group configuration includes the following required attributes:

- name -- the name of the server group
- profile -- the name of the profile the servers in the group should run

In addition, the following optional elements are available:

- socket-binding-group -- specifies the name of the default socket binding group to use on servers in the group. Can be overridden on a per-server basis in `host.xml`. If not provided in the `server-group` element, it must be provided for each server in `host.xml`.
- deployments -- the deployment content that should be deployed on the servers in the group.
- deployment-overlays -- the overlays and their associated deployments.
- system-properties -- system properties that should be set on all servers in the group
- jvm -- default jvm settings for all servers in the group. The Host Controller will merge these settings with any provided in `host.xml` to derive the settings to use to launch the server's JVM. See JVM settings for further details.

Server

Each "Server" in the above diagram represents an actual application server instance. The server runs in a separate JVM process from the Host Controller. The Host Controller is responsible for launching that process. (In a managed domain the end user cannot directly launch a server process from the command line.)

The Host Controller synthesizes the server's configuration by combining elements from the domain wide configuration (from `domain.xml`) and the host-specific configuration (from `host.xml`).

Deciding between running standalone servers or a managed domain

Which use cases are appropriate for managed domain and which are appropriate for standalone servers? A managed domain is all about coordinated multi-server management -- with it WildFly provides a central point through which users can manage multiple servers, with rich capabilities to keep those servers' configurations consistent and the ability to roll out configuration changes (including deployments) to the servers in a coordinated fashion.

It's important to understand that the choice between a managed domain and standalone servers is all about how your servers are managed, not what capabilities they have to service end user requests. This distinction is particularly important when it comes to high availability clusters. It's important to understand that HA functionality is orthogonal to running standalone servers or a managed domain. That is, a group of standalone servers can be configured to form an HA cluster. The domain and standalone modes determine how the servers are managed, not what capabilities they provide.

So, given all that:

- A single server installation gains nothing from running in a managed domain, so running a standalone server is a better choice.
- For multi-server production environments, the choice of running a managed domain versus standalone servers comes down to whether the user wants to use the centralized management capabilities a managed domain provides. Some enterprises have developed their own sophisticated multi-server management capabilities and are comfortable coordinating changes across a number of independent WildFly instances. For these enterprises, a multi-server architecture comprised of individual standalone servers is a good option.
- Running a standalone server is better suited for most development scenarios. Any individual server configuration that can be achieved in a managed domain can also be achieved in a standalone server, so even if the application being developed will eventually run in production on a managed domain installation, much (probably most) development can be done using a standalone server.
- Running a managed domain mode can be helpful in some advanced development scenarios; i.e. those involving interaction between multiple WildFly instances. Developers may find that setting up various servers as members of a domain is an efficient way to launch a multi-server cluster.

5.15 Domain Setup

To run a group of servers as a managed domain you need to configure both the domain controller and each host that joins the domain. This sections focuses on the network configuration for the domain and host controller components. For background information users are encouraged to review the Operating modes and Configuration Files sections.

5.15.1 Domain Controller Configuration

The domain controller is the central government for a managed domain. A domain controller configuration requires two steps:

- A host needs to be configured to act as the Domain Controller for the whole domain
- The host must expose an addressable management interface binding for the managed hosts to communicate with it

Example IP Addresses

In this example the domain controller uses 192.168.0.101 and the host controller 192.168.0.10

Configuring a host to act as the Domain Controller is done through the `domain-controller` declaration in `host.xml`. If it includes the `<local/>` element, then this host will become the domain controller:

```
<domain-controller>
   <local/>
</domain-controller>
```

(See domain/configuration/host.xml)

A host acting as the Domain Controller *must* expose a management interface on an address accessible to the other hosts in the domain. Exposing an HTTP(S) management interface is not required, but is recommended as it allows the Administration Console to work:

```
<management-interfaces>
  <native-interface security-realm="ManagementRealm">
    <socket interface="management" port="${jboss.management.native.port:9999}"/>
  </native-interface>
  <http-interface security-realm="ManagementRealm">
    <socket interface="management" port="${jboss.management.http.port:9990}"/>
  </http-interface>
</management-interfaces>
```

The interface attributes above refer to a named interface declaration later in the host.xml file. This interface declaration will be used to resolve a corresponding network interface.

```
<interfaces>
   <interface name="management">
       <inet-address value="192.168.0.101"/>
   </interface>
</interfaces>
```

(See domain/configuration/host.xml)

Please consult the chapter "Interface Configuration" for a more detailed explanation on how to configure network interfaces.

Next by default the master domain controller is configured to require authentication so a user needs to be added that can be used by the slave domain controller to connect.

Make use of the `add-user` utility to add a new user, for this example I am adding a new user called slave.

`add-user` MUST be run on the master domain controller and NOT the slave.

When you reach the final question of the interactive flow answer `y` or `yes` to indicate that the new user will be used for a process e.g.

```
Is this new user going to be used for one AS process to connect to another AS process e.g. slave
domain controller?
yes/no? y
To represent the user add the following to the server-identities definition <secret
value="cE3EBEkE=" />
```

Make a note of the XML Element output as that is going to be required within the slave configuration.

5.15.2 Host Controller Configuration

Once the domain controller is configured correctly you can proceed with any host that should join the domain. The host controller configuration requires three steps:

- The logical host name (within the domain) needs to be distinct
- The host controller needs to know the domain controller IP address

Provide a distinct, logical name for the host. In the following example we simply name it "slave":

```
<host xmlns="urn:jboss:domain:3.0"
     name="slave">
[...]
</host>
```

(See domain/configuration/host.xml)

If the `name` attribute is not set, the default name for the host will be the value of the `jboss.host.name` system property. If that is not set, the value of the `HOSTNAME` or `COMPUTERNAME` environment variable will be used, one of which will be set on most operating systems. If neither is set the name will be the value of `InetAddress.getLocalHost().getHostName()`.

A security realm needs to be defined to hold the identity of the slave. Since it is performing a specific purpose I would suggest a new realm is defined although it is possible to combine this with an existing realm.

```
<security-realm name="SlaveRealm">
  <server-identities>
    <secret value="cE3EBEkE=" />
  </server-identities>
</security-realm>
```

The `<secret />` element here is the one output from `add-user` previously. To create the `<secret />` element yourself the `value` needs to be the password encoded using Base64.

Tell it how to find the domain controller so it can register itself with the domain:

```
<domain-controller>
   <remote protocol="remote" host="192.168.0.101" port="9999" username="slave"
security-realm="SlaveRealm"/>
</domain-controller>
```

Since we have also exposed the HTTP management interface we could also use :

```
<domain-controller>
   <remote protocol="http-remoting" host="192.168.0.101" port="9990" username="slave"
security-realm="SlaveRealm"/>
</domain-controller>
```

(See domain/configuration/host.xml)

The username attribute here is optional, if it is omitted then the name of the host will be used instead, in this example that was already set to name.

The name of each host needs to be unique when registering with the domain controller, however the username does not - using the username attribute allows the same account to be used by multiple hosts if this makes sense in your environment.

The `<remote />` element is also associated with the security realm `SlaveRealm`, this is how it picks up the password from the `<secret />` element.

Ignoring domain wide resources

WildFly 10 and later make it easy for slave host controllers to "ignore" parts of the domain wide configuration. What does the mean and why is it useful?

One of the responsibilities of the Domain Controller is ensuring that all running Host Controllers have a consistent local copy of the domain wide configuration (i.e. those resources whose address does not begin with `/host=*`, i.e. those that are persisted in `domain.xml`. Having that local copy allows a user to do the following things:

- Ask the slave to launch its already configured servers, even if the Domain Controller is not running.
- Configured new servers, using different server groups from those current running, and ask the slave to launch them, even if the Domain Controller is not running.
- Reconfigure the slave to act as the Domain Controller, allowing it to take over as the master if the previous master has failed or been shut down.

However, of these three things only the latter two require that the slave maintain a *complete* copy of the domain wide configuration. The first only requires the slave to have the *portion* of the domain wide configuration that is relevant to its current servers. And the first use case is the most common one. A slave that is only meant to support the first use case can safely "ignore" portions of the domain wide configuration. And there are benefits to ignoring some resources:

- If a server group is ignored, and the deployments mapped to that server group aren't mapped to other non-ignored groups, then the slave does not need to pull down a copy of the deployment content from the master. That can save disk space on the slave, improve the speed of starting new hosts and reduce network traffic.
- WildFly supports "mixed domains" where a later version Domain Controller can manage slaves running previous versions. But those "legacy" slaves cannot understand configuration resources, attributes and operations introduced in newer versions. So any attempt to use newer things in the domain wide configuration will fail unless the legacy slaves are ignoring the relevant resources. But ignoring resources will allow the legacy slaves to work fine managing servers using profiles without new concepts, while other hosts can run servers with profiles that take advantage of the latest features.

Prior to WildFly 10, a slave could be configured to ignore some resources, but the mechanism was not particularly user friendly:

- The resources to be ignored had to be listed in a fair amount of detail in each host's configuration.
- If a new resource is added and needs to be ignored, then **each** host that needs to ignore that must be updated to record that.

Starting with WildFly 10, this kind of detailed configuration is no longer required. Instead, with the standard versions of `host.xml`, the slave will behave as follows:

- If the slave was started with the `--backup` command line parameter, the behavior will be the same as releases prior to 10; i.e. only resources specifically configured to be ignored will be ignored.
- Otherwise, the slave will "ignore unused resources".

What does "ignoring unused resources" mean?

- Any server-group that is not referenced by one of the host's server-config resources is ignored.
- Any profile that is not referenced by a non-ignored server-group, either directly or indirectly via the profile resource's 'include' attribute, is ignored
- Any socket-binding-group that is not directly referenced by one of the host's server-config resources, or referenced by a non-ignored server-group, is ignored
- Extension resources will not be automatically ignored, even if no non-ignored profile uses the extension. Ignoring an extension requires explicit configuration. Perhaps in a future release extensions will be explicitly ignored.
- If a change is made to the slave host's configuration or to the domain wide configuration that reduces the set of ignored resources, then as part of handling that change the slave will contact the master to pull down the missing pieces of configuration and will integrate those pieces in its local copy of the management model. Examples of such changes include adding a new server-config that references a previously ignored server-group or socket-binding-group, changing the server-group or socket-binding-group assigned to a server-config, changing the profile or socket-binding-group assigned to a non-ignored server-group, or adding a profile or socket-binding-group to the set of those included directly or indirectly by a non-ignored profile or socket-binding-group.

The default behavior can be changed, either to always ignore unused resources, even if `--backup` is used, or to not ignore unused resources, by updating the domain-controller element in the `host-xml` file and setting the `ignore-unused-configuration` attribute:

```
<domain-controller>
        <remote security-realm="ManagementRealm" ignore-unused-configuration="false">
            <discovery-options>
                <static-discovery name="primary"
protocol="${jboss.domain.master.protocol:remote}" host="${jboss.domain.master.address}"
port="${jboss.domain.master.port:9999}"/>
            </discovery-options>
        </remote>
    </domain-controller>
```

The "ignore unused resources" behavior can be used in combination with the pre-WildFly 10 detailed specification of what to ignore. If that is done both the unused resources and the explicitly declared resources will be ignored. Here's an example of such a configuration, one where the slave cannot use the "org.example.foo" extension that has been installed on the Domain Controller and on some slaves, but not this one:

```
<domain-controller>
        <remote security-realm="ManagementRealm" ignore-unused-configuration="true">
            <ignored-resources type="extension">
                <instance name="org.example.foo"/>
            </ignored-resources>
            <discovery-options>
                <static-discovery name="primary"
protocol="${jboss.domain.master.protocol:remote}" host="${jboss.domain.master.address}"
port="${jboss.domain.master.port:9999}"/>
            </discovery-options>
        </remote>
    </domain-controller>
```

5.15.3 Server groups

The domain controller defines one or more server groups and associates each of these with a profile and a socket binding group, and also :

```
<server-groups>
    <server-group name="main-server-group" profile="default">
        <jvm name="default">
           <heap size="64m" max-size="512m"/>
           <permgen size="128m" max-size="128m"/>
        </jvm>
        <socket-binding-group ref="standard-sockets"/>
    </server-group>
    <server-group name="other-server-group" profile="bigger">
        <jvm name="default">
            <heap size="64m" max-size="512m"/>
        </jvm>
        <socket-binding-group ref="bigger-sockets"/>
    </server-group>
</server-groups>
```

(See domain/configuration/domain.xml)

The domain controller also defines the socket binding groups and the profiles. The socket binding groups define the default socket bindings that are used:

```
<socket-binding-groups>
    <socket-binding-group name="standard-sockets" default-interface="public">
        <socket-binding name="http" port="8080"/>
        [...]
    </socket-binding-group>
    <socket-binding-group name="bigger-sockets" include="standard-sockets"
default-interface="public">
        <socket-binding name="unique-to-bigger" port="8123"/>
    </socket-binding-group>
</socket-binding-groups>
```

(See domain/configuration/domain.xml)

In this example the `bigger-sockets` group includes all the socket bindings defined in the `standard-sockets` groups and then defines an extra socket binding of its own.

A profile is a collection of subsystems, and these subsystems are what implement the functionality people expect of an application server.

```
<profiles>
    <profile name="default">
        <subsystem xmlns="urn:jboss:domain:web:1.0">
            <connector name="http" scheme="http" protocol="HTTP/1.1" socket-binding="http"/>
            [...]
        </subsystem>
        <\!-\- The rest of the subsystems here \-->
        [...]
    </profile>
    <profile name="bigger">
        <subsystem xmlns="urn:jboss:domain:web:1.0">
            <connector name="http" scheme="http" protocol="HTTP/1.1" socket-binding="http"/>
            [...]
        </subsystem>
        <\!-\- The same subsystems as defined by 'default' here \-->
        [...]
        <subsystem xmlns="urn:jboss:domain:fictional-example:1.0">
            <socket-to-use name="unique-to-bigger"/>
        </subsystem>
    </profile>
</profiles>
```

(See domain/configuration/domain.xml)

Here we have two profiles. The `bigger` profile contains all the same subsystems as the `default` profile (athough the parameters for the various subsystems could be different in each profile), and adds the `fictional-example` subsystem which references the `unique-to-bigger` socket binding.

5.15.4 Servers

The host controller defines one or more servers:

```
<servers>
    <server name="server-one" group="main-server-group">
        <\!-\- server-one inherits the default socket-group declared in the server-group \-->
        <jvm name="default"/>
    </server>

    <server name="server-two" group="main-server-group" auto-start="true">
        <socket-binding-group ref="standard-sockets" port-offset="150"/>
        <jvm name="default">
            <heap size="64m" max-size="256m"/>
        </jvm>
    </server>

    <server name="server-three" group="other-server-group" auto-start="false">
        <socket-binding-group ref="bigger-sockets" port-offset="250"/>
    </server>
</servers>
```

(See domain/configuration/host.xml)

`server-one` and `server-two` both are associated with `main-server-group` so that means they both run the subsystems defined by the `default` profile, and have the socket bindings defined by the `standard-sockets` socket binding group. Since all the servers defined by a host will be run on the same physical host we would get port conflicts unless we used `<socket-binding-group ref="standard-sockets" port-offset="150"/>` for `server-two`. This means that `server-two` will use the socket bindings defined by `standard-sockets` but it will add `150` to each port number defined, so the value used for `http` will be `8230` for `server-two`.

`server-three` will not be started due to its `auto-start="false"`. The default value if no `auto-start` is given is `true` so both `server-one` and `server-two` will be started when the host controller is started. `server-three` belongs to `other-server-group`, so if its `auto-start` were changed to `true` it would start up using the subsystems from the `bigger` profile, and it would use the `bigger-sockets` socket binding group.

JVM

The host controller contains the main `jvm` definitions with arguments:

```
<jvms>
    <jvm name="default">
        <heap size="64m" max-size="128m"/>
    </jvm>
</jvms>
```

(See domain/configuration/host.xml)
From the preceeding examples we can see that we also had a `jvm` reference at server group level in the domain controller. The jvm's name **must** match one of the definitions in the host controller. The values supplied at domain controller and host controller level are combined, with the host controller taking precedence if the same parameter is given in both places.

Finally, as seen, we can also override the `jvm` at server level. Again, the jvm's name **must** match one of the definitions in the host controller. The values are combined with the ones coming in from domain controller and host controller level, this time the server definition takes precedence if the same parameter is given in all places.

Following these rules the jvm parameters to start each server would be

Server	JVM parameters
`server-one`	`-Xms64m -Xmx128m`
`server-two`	`-Xms64m -Xmx256m`
`server-three`	`-Xms64m -Xmx128m`

5.16 Interfaces and ports

5.16.1 Interface declarations

WildFly uses named interface references throughout the configuration. A network interface is declared by specifying a logical name and a selection criteria for the physical interface:

```
[standalone@localhost:9990 /] :read-children-names(child-type=interface)
{
   "outcome" => "success",
   "result" => [
       "management",
       "public"
   ]
}
```

This means the server in question declares two interfaces: One is referred to as "*management*"; the other one "*public*". The "*management*" interface is used for all components and services that are required by the management layer (i.e. the HTTP Management Endpoint). The "*public*" interface binding is used for any application related network communication (i.e. Web, Messaging, etc). There is nothing special about these names; interfaces can be declared with any name. Other sections of the configuration can then reference those interfaces by their logical name, rather than having to include the full details of the interface (which, on servers in a management domain, may vary on different machines).

The `domain.xml`, `host.xml` and `standalone.xml` configuration files all include a section where interfaces can be declared. If we take a look at the XML declaration it reveals the selection criteria. The criteria is one of two types: either a single element indicating that the interface should be bound to a wildcard address, or a set of one or more characteristics that an interface or address must have in order to be a valid match. The selection criteria in this example are specific IP addresses for each interface:

```
<interfaces>
  <interface name="management">
   <inet-address value="127.0.0.1"/>
  </interface>
  <interface name="public">
   <inet-address value="127.0.0.1"/>
  </interface>
</interfaces>
```

Some other examples:

```
<interface name="global">
   <!-- Use the wildcard address -->
   <any-address/>
</interface>

<interface name="external">
   <nic name="eth0"/>
</interface>

<interface name="default">
   <!-- Match any interface/address on the right subnet if it's
        up, supports multicast and isn't point-to-point -->
   <subnet-match value="192.168.0.0/16"/>
   <up/>
   <multicast/>
   <not>
      <point-to-point/>
   </not>
</interface>
```

The -b command line argument

WildFly supports using the `-b` command line argument to specify the address to assign to interfaces. See Controlling the Bind Address with -b for further details.

5.16.2 Socket Binding Groups

The socket configuration in WildFly works similarly to the interfaces declarations. Sockets are declared using a logical name, by which they will be referenced throughout the configuration. Socket declarations are grouped under a certain name. This allows you to easily reference a particular socket binding group when configuring server groups in a managed domain. Socket binding groups reference an interface by its logical name:

```
<socket-binding-group name="standard-sockets" default-interface="public">
  <socket-binding name="management-http" interface="management"
port="${jboss.management.http.port:9990}"/>
  <socket-binding name="management-https" interface="management"
port="${jboss.management.https.port:9993}"/>
  <socket-binding name="ajp" port="${jboss.ajp.port:8009}"/>
  <socket-binding name="http" port="${jboss.http.port:8080}"/>
  <socket-binding name="https" port="${jboss.https.port:8443}"/>
  <socket-binding name="txn-recovery-environment" port="4712"/>
  <socket-binding name="txn-status-manager" port="4713"/>
</socket-binding-group>
```

A socket binding includes the following information:

- name -- logical name of the socket configuration that should be used elsewhere in the configuration
- port -- base port to which a socket based on this configuration should be bound. (Note that servers can be configured to override this base value by applying an increment or decrement to all port values.)
- interface (optional) -- logical name (see "Interfaces declarations" above) of the interface to which a socket based on this configuration should be bound. If not defined, the value of the "default-interface" attribute from the enclosing socket binding group will be used.
- multicast-address (optional) -- if the socket will be used for multicast, the multicast address to use
- multicast-port (optional) -- if the socket will be used for multicast, the multicast port to use
- fixed-port (optional, defaults to false) -- if true, declares that the value of port should always be used for the socket and should not be overridden by applying an increment or decrement

5.16.3 IPv4 versus IPv6

WildFly supports the use of both IPv4 and IPv6 addresses. By default, WildFly is configured for use in an IPv4 network and so if you are running in an IPv4 network, no changes are required. If you need to run in an IPv6 network, the changes required are minimal and involve changing the JVM stack and address preferences, and adjusting any interface IP address values specified in the configuration (standalone.xml or domain.xml).

Stack and address preference

The system properties java.net.preferIPv4Stack and java.net.preferIPv6Addresses are used to configure the JVM for use with IPv4 or IPv6 addresses. With WildFly, in order to run using IPv4 addresses, you need to specify java.net.preferIPv4Stack=true; in order to run with IPv6 addresses, you need to specify java.net.preferIPv4Stack=false (the JVM default) and java.net.preferIPv6Addresses=true. The latter ensures that any hostname to IP address conversions always return IPv6 address variants.

These system properties are conveniently set by the JAVA_OPTS environment variable, defined in the standalone.conf (or domain.conf) file. For example, to change the IP stack preference from its default of IPv4 to IPv6, edit the standalone.conf (or domain.conf) file and change its default IPv4 setting:

```
if [ "x$JAVA_OPTS" = "x" ]; then
   JAVA_OPTS=" ... -Djava.net.preferIPv4Stack=true ..."
...
```

to an IPv6 suitable setting:

```
if [ "x$JAVA_OPTS" = "x" ]; then
   JAVA_OPTS=" ... -Djava.net.preferIPv4Stack=false -Djava.net.preferIPv6Addresses=true ..."
...
```

IP address literals

To change the IP address literals referenced in standalone.xml (or domain.xml), first visit the interface declarations and ensure that valid IPv6 addresses are being used as interface values. For example, to change the default configuration in which the loopback interface is used as the primary interface, change from the IPv4 loopback address:

```
<interfaces>
  <interface name="management">
    <inet-address value="${jboss.bind.address.management:127.0.0.1}"/>
  </interface>
  <interface name="public">
    <inet-address value="${jboss.bind.address:127.0.0.1}"/>
  </interface>
</interfaces>
```

to the IPv6 loopback address:

```
<interfaces>
  <interface name="management">
    <inet-address value="${jboss.bind.address.management:[::1]}"/>
  </interface>
  <interface name="public">
    <inet-address value="${jboss.bind.address:[::1]}"/>
  </interface>
</interfaces>
```

Note that when embedding IPv6 address literals in the substitution expression, square brackets surrounding the IP address literal are used to avoid ambiguity. This follows the convention for the use of IPv6 literals in URLs.

Over and above making such changes for the interface definitions, you should also check the rest of your configuration file and adjust IP address literals from IPv4 to IPv6 as required.

5.17 Management API reference

This section is an in depth reference to the WildFly management API. Readers are encouraged to read the Management Clients and Core management concepts sections for fundamental background information, as well as the Management tasks and Domain Setup sections for key task oriented information. This section is meant as an in depth reference to delve into some of the key details.

5.17.1 Global operations

The WildFly management API includes a number of operations that apply to every resource.

The read-resource operation

Reads a management resource's attribute values along with either basic or complete information about any child resources. Supports the
following parameters, none of which are required:

- `recursive` – (boolean, default is `false`) – whether to include complete information about child resources, recursively.
- `recursive-depth` – (int) – The depth to which information about child resources should be included if `recursive` is `true`. If not set, the depth will be unlimited; i.e. all descendant resources will be included.
- `proxies` – (boolean, default is `false`) – whether to include remote resources in a recursive query (i.e. host level resources from slave Host Controllers in a query of the Domain Controller; running server resources in a query of a host).
- `include-runtime` – (boolean, default is `false`) – whether to include runtime attributes (i.e. those whose value does not come from the persistent configuration) in the response.
- `include-defaults` – (boolean, default is `true`) – whether to include in the result default values not set by users. Many attributes have a default value that will be used in the runtime if the users have not provided an explicit value. If this parameter is `false` the value for such attributes in the result will be `undefined`. If `true` the result will include the default value for such parameters.

The read-attribute operation

Reads the value of an individual attribute. Takes a single, required, parameter:

- `name` – (string) – the name of the attribute to read.
- `include-defaults` – (boolean, default is `true`) – whether to include in the result default values not set by users. Many attributes have a default value that will be used in the runtime if the users have not provided an explicit value. If this parameter is `false` the value for such attributes in the result will be `undefined`. If `true` the result will include the default value for such parameters.

The write-attribute operation

Writes the value of an individual attribute. Takes two required parameters:

- `name` – (string) – the name of the attribute to write.
- `value` – (type depends on the attribute being written) – the new value.

The undefine-attribute operation

Sets the value of an individual attribute to the `undefined` value, if such a value is allowed for the attribute. The operation will fail if the `undefined` value is not allowed. Takes a single required parameter:

- `name` – (string) – the name of the attribute to write.

The list-add operation

Adds an element to the value of a list attribute, adding the element to the end of the list unless the optional attribute `index` is passed:

- `name` – (string) – the name of the list attribute to add new value to.
- `value` – (type depends on the element being written) – the new element to be added to the attribute value.
- `index` – (int, optional) – index where in the list to add the new element. By default it is `undefined` meaning add at the end. Index is zero based.

This operation will fail if the specified attribute is not a list.

The list-remove operation

Removes an element from the value of a list attribute, either the element at a specified `index`, or the first element whose value matches a specified `value`:

- `name` – (string) – the name of the list attribute to add new value to.
- `value` – (type depends on the element being written, optional) – the element to be removed. Optional and ignored if `index` is specified.
- `index` – (int, optional) – index in the list whose element should be removed. By default it is `undefined`, meaning `value` should be specified.

This operation will fail if the specified attribute is not a list.

The list-get operation

Gets one element from a list attribute by its index

- `name` – (string) – the name of the list attribute
- `index` – (int, required) – index of element to get from list

This operation will fail if the specified attribute is not a list.

The list-clear operation

Empties the list attribute. It is different from `:undefine-attribute` as it results in attribute of type list with 0 elements, whereas `:undefine-attribute` results in an `undefined` value for the attribute

- `name` – (string) – the name of the list attribute

This operation will fail if the specified attribute is not a list.

The map-put operation

Adds an key/value pair entry to the value of a map attribute:

- `name` – (string) – the name of the map attribute to add the new entry to.
- `key` – (string) – the key of the new entry to be added.
- `value` – (type depends on the entry being written) – the value of the new entry to be added to the attribute value.

This operation will fail if the specified attribute is not a map.

The map-remove operation

Removes an entry from the value of a map attribute:

- `name` – (string) – the name of the map attribute to remove the new entry from.
- `key` – (string) – the key of the entry to be removed.

This operation will fail if the specified attribute is not a map.

The map-get operation

Gets the value of one entry from a map attribute

- `name` – (string) – the name of the map attribute
- `key` – (string) – the key of the entry.

This operation will fail if the specified attribute is not a map.

The map-clear operation

Empties the map attribute. It is different from `:undefine-attribute` as it results in attribute of type map with 0 entries, whereas `:undefine-attribute` results in an `undefined` value for the attribute

- `name` – (string) – the name of the map attribute

This operation will fail if the specified attribute is not a map.

The read-resource-description operation

Returns the description of a resource's attributes, types of children and, optionally, operations. Supports the following parameters, none of which are required:

- `recursive` – (boolean, default is `false`) – whether to include information about child resources, recursively.
- `proxies` – (boolean, default is `false`) – whether to include remote resources in a recursive query (i.e. host level resources from slave Host Controllers in a query of the Domain Controller; running server resources in a query of a host)
- `operations` – (boolean, default is `false`) – whether to include descriptions of the resource's operations
- `inherited` – (boolean, default is `true`) – if `operations` is `true`, whether to include descriptions of operations inherited from higher level resources. The global operations described in this section are themselves inherited from the root resource, so the primary effect of setting `inherited` to `false` is to exclude the descriptions of the global operations from the output.

See Description of the Management Model for details on the result of this operation.

The read-operation-names operation

Returns a list of the names of all the operations the resource supports. Takes no parameters.

The read-operation-description operation

Returns the description of an operation, along with details of its parameter types and its return value. Takes a single, required, parameter:

- `name` – (string) – the name of the operation

See Description of the Management Model for details on the result of this operation.

The read-children-types operation

Returns a list of the types of child resources the resource supports. Takes two optional parameters:

- `include-aliases` – (boolean, default is `false`) – whether to include alias children (i.e. those which are aliases of other sub-resources) in the response.
- `include-singletons` – (boolean, default is `false`) – whether to include singleton children (i.e. those are children that acts as resource aggregate and are registered with a wildcard name) in the response wildfly-dev discussion around this topic.

The read-children-names operation

Returns a list of the names of all child resources of a given type. Takes a single, required, parameter:

- `child-type` – (string) – the name of the type

The read-children-resources operation

Returns information about all of a resource's children that are of a given type. For each child resource, the returned information is equivalent to executing the `read-resource` operation on that resource. Takes the following parameters, of which only {{child-type} is required:

- `child-type` – (string) – the name of the type of child resource
- `recursive` – (boolean, default is `false`) – whether to include complete information about child resources, recursively.
- `recursive-depth` – (int) – The depth to which information about child resources should be included if `recursive` is {{true}. If not set, the depth will be unlimited; i.e. all descendant resources will be included.
- `proxies` – (boolean, default is `false`) – whether to include remote resources in a recursive query (i.e. host level resources from slave Host Controllers in a query of the Domain Controller; running server resources in a query of a host)
- `include-runtime` – (boolean, default is `false`) – whether to include runtime attributes (i.e. those whose value does not come from the persistent configuration) in the response.
- `include-defaults` – (boolean, default is `true`) – whether to include in the result default values not set by users. Many attributes have a default value that will be used in the runtime if the users have not provided an explicit value. If this parameter is `false` the value for such attributes in the result will be `undefined`. If `true` the result will include the default value for such parameters.

The read-attribute-group operation

Returns a list of attributes of a type for a given attribute group name. For each attribute the returned information is equivalent to executing the `read-attribute` operation of that resource. Takes the following parameters, of which only {{name} is required:

- `name` – (string) – the name of the attribute group to read.
- `include-defaults` – (boolean, default is `true`) – whether to include in the result default values not set by users. Many attributes have a default value that will be used in the runtime if the users have not provided an explicit value. If this parameter is `false` the value for such attributes in the result will be `undefined`. If `true` the result will include the default value for such parameters.
- `include-runtime` – (boolean, default is `false`) – whether to include runtime attributes (i.e. those whose value does not come from the persistent configuration) in the response.
- `include-aliases` – (boolean, default is `false`) – whether to include alias attributes (i.e. those which are alias of other attributes) in the response.

The read-attribute-group-names operation

Returns a list of attribute groups names for a given type. Takes no parameters.

Standard Operations

Besides the global operations described above, by convention nearly every resource should expose an `add` operation and a `remove` operation. Exceptions to this convention are the root resource, and resources that do not store persistent configuration and are created dynamically at runtime (e.g. resources representing the JVM's platform mbeans or resources representing aspects of the running state of a deployment.)

The add operation

The operation that creates a new resource must be named `add`. The operation may take zero or more parameters; what those parameters are depends on the resource being created.

The remove operation

The operation that removes an existing resource must be named `remove`. The operation should take no parameters.

5.17.2 Detyped management and the jboss-dmr library

The management model exposed by WildFly is very large and complex. There are dozens, probably hundreds of logical concepts involved – hosts, server groups, servers, subsystems, datasources, web connectors, and on and on – each of which in a classic objected oriented API design could be represented by a Java *type* (i.e. a Java class or interface.) However, a primary goal in the development of WildFly's native management API was to ensure that clients built to use the API had as few compile-time and run-time dependencies on JBoss-provided classes as possible, and that the API exposed by those libraries be powerful but also simple and stable. A management client running with the management libraries created for an earlier version of WildFly should still work if used to manage a later version domain. The management client libraries needed to be *forward compatible*.

It is highly unlikely that an API that consists of hundreds of Java types could be kept forward compatible. Instead, the WildFly management API is a *detyped* API. A detyped API is like decaffeinated coffee – it still has a little bit of caffeine, but not enough to keep you awake at night. WildFly's management API still has a few Java types in it (it's impossible for a Java library to have no types!) but not enough to keep you (or us) up at night worrying that your management clients won't be forward compatible.

A detyped API works by making it possible to build up arbitrarily complex data structures using a small number of Java types. All of the parameter values and return values in the API are expressed using those few types. Ideally, most of the types are basic JDK types, like `java.lang.String`, `java.lang.Integer`, etc. In addition to the basic JDK types, WildFly's detyped management API uses a small library called **jboss-dmr**. The purpose of this section is to provide a basic overview of the jboss-dmr library.

Even if you don't use jboss-dmr directly (probably the case for all but a few users), some of the information in this section may be useful. When you invoke operations using the application server's Command Line Interface, the return values are just the text representation of of a jboss-dmr `ModelNode`. If your CLI commands require complex parameter values, you may yourself end up writing the text representation of a `ModelNode`. And if you use the HTTP management API, all response bodies as well as the request body for any POST will be a JSON representation of a `ModelNode`.

The source code for jboss-dmr is available on Github. The maven coordinates for a jboss-dmr release are `org.jboss.jboss-dmr:jboss-dmr`.

ModelNode and ModelType

The public API exposed by jboss-dmr is very simple: just three classes, one of which is an enum!

The primary class is `org.jboss.dmr.ModelNode`. A `ModelNode` is essentially just a wrapper around some *value*; the value is typically some basic JDK type. A `ModelNode` exposes a `getType()` method. This method returns a value of type `org.jboss.dmr.ModelType`, which is an enum of all the valid types of values. And that's 95% of the public API; a class and an enum. (We'll get to the third class, `Property`, below.)

Basic ModelNode manipulation

To illustrate how to work with `ModelNode` s, we'll use the Beanshell scripting library. We won't get into many details of beanshell here; it's a simple and intuitive tool and hopefully the following examples are as well.

We'll start by launching a beanshell interpreter, with the jboss-dmr library available on the classpath. Then we'll tell beanshell to import all the jboss-dmr classes so they are available for use:

```
$ java -cp bsh-2.0b4.jar:jboss-dmr-1.0.0.Final.jar bsh.Interpreter
BeanShell 2.0b4 - by Pat Niemeyer (pat@pat.net)
bsh % import org.jboss.dmr.*;
bsh %
```

Next, create a `ModelNode` and use the beanshell `print` function to output what type it is:

```
bsh % ModelNode node = new ModelNode();
bsh % print(node.getType());
UNDEFINED
```

A new `ModelNode` has no value stored, so its type is `ModelType.UNDEFINED`.

Use one of the overloaded `set` method variants to assign a node's value:

```
bsh % node.set(1);
bsh % print(node.getType());
INT
bsh % node.set(true);
bsh % print(node.getType());
BOOLEAN
bsh % node.set("Hello, world");
bsh % print(node.getType());
STRING
```

Use one of the `asXXX()` methods to retrieve the value:

```
bsh % node.set(2);
bsh % print(node.asInt());
2
bsh % node.set("A string");
bsh % print(node.asString());
A string
```

`ModelNode` will attempt to perform type conversions when you invoke the `asXXX` methods:

```
bsh % node.set(1);
bsh % print(node.asString());
1
bsh % print(node.asBoolean());
true
bsh % node.set(0);
bsh % print(node.asBoolean());
false
bsh % node.set("true");
bsh % print(node.asBoolean());
true
```

Not all type conversions are possible:

```
bsh % node.set("A string");
bsh % print(node.asInt());
// Error: // Uncaught Exception: Method Invocation node.asInt : at Line: 20 : in file: <unknown
file> : node .asInt ( )

Target exception: java.lang.NumberFormatException: For input string: "A string"

java.lang.NumberFormatException: For input string: "A string"
 at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
 at java.lang.Integer.parseInt(Integer.java:449)
 at java.lang.Integer.parseInt(Integer.java:499)
 at org.jboss.dmr.StringModelValue.asInt(StringModelValue.java:61)
 at org.jboss.dmr.ModelNode.asInt(ModelNode.java:117)
        ....
```

The `ModelNode.getType()` method can be used to ensure a node has an expected value type before attempting a type conversion.

One `set` variant takes another `ModelNode` as its argument. The value of the passed in node is copied, so there is no shared state between the two model nodes:

```
bsh % node.set("A string");
bsh % ModelNode another = new ModelNode();
bsh % another.set(node);
bsh % print(another.asString());
A string
bsh % node.set("changed");
bsh % print(node.asString());
changed
bsh % print(another.asString());
A string
```

A `ModelNode` can be cloned. Again, there is no shared state between the original node and its clone:

```
bsh % ModelNode clone = another.clone();
bsh % print(clone.asString());
A string
bsh % another.set(42);
bsh % print(another.asString());
42
bsh % print(clone.asString());
A string
```

Use the `protect()` method to make a `ModelNode` immutable:

```
bsh % clone.protect();
bsh % clone.set("A different string");
// Error: // Uncaught Exception: Method Invocation clone.set : at Line: 15 : in file: <unknown
file> : clone .set ( "A different string" )

Target exception: java.lang.UnsupportedOperationException

java.lang.UnsupportedOperationException
 at org.jboss.dmr.ModelNode.checkProtect(ModelNode.java:1441)
 at org.jboss.dmr.ModelNode.set(ModelNode.java:351)
        ....
```

Lists

The above examples aren't particularly interesting; if all we can do with a `ModelNode` is wrap a simple Java primitive, what use is that? However, a `ModelNode`'s value can be more complex than a simple primitive, and using these more complex types we can build complex data structures. The first more complex type is `ModelType.LIST`.

Use the `add` methods to initialize a node's value as a list and add to the list:

```
bsh % ModelNode list = new ModelNode();
bsh % list.add(5);
bsh % list.add(10);
bsh % print(list.getType());
LIST
```

Use `asInt()` to find the size of the list:

```
bsh % print(list.asInt());
2
```

Use the overloaded `get` method variant that takes an int param to retrieve an item. The item is returned as a `ModelNode`:

```
bsh % ModelNode child = list.get(1);
bsh % print(child.asInt());
10
```

Elements in a list need not all be of the same type:

```
bsh % list.add("A string");
bsh % print(list.get(1).getType());
INT
bsh % print(list.get(2).getType());
STRING
```

Here's one of the trickiest things about jboss-dmr: *The* `get` *methods actually mutate state; they are not "read-only"*. For example, calling `get` with an index that does not exist yet in the list will actually create a child of type `ModelType.UNDEFINED` at that index (and will create UNDEFINED children for any intervening indices.)

```
bsh % ModelNode four = list.get(4);
bsh % print(four.getType());
UNDEFINED
bsh % print(list.asInt());
6
```

Since the `get` call always returns a `ModelNode` and never `null` it is safe to manipulate the return value:

```
bsh % list.get(5).set(30);
bsh % print(list.get(5).asInt());
30
```

That's not so interesting in the above example, but later on with node of type `ModelType.OBJECT` we'll see how that kind of method chaining can let you build up fairly complex data structures with a minimum of code.

Use the `asList()` method to get a `List<ModelNode>` of the children:

```
bsh % for (ModelNode element : list.asList()) {
print(element.getType());
}
INT
INT
STRING
UNDEFINED
UNDEFINED
INT
```

The `asString()` and `toString()` methods provide slightly differently formatted text representations of a `ModelType.LIST` node:

```
bsh % print(list.asString());
[5,10,"A string",undefined,undefined,30]
bsh % print(list.toString());
[
    5,
    10,
    "A string",
    undefined,
    undefined,
    30
]
```

Finally, if you've previously used `set` to assign a node's value to some non-list type, you cannot use the `add` method:

```
bsh % node.add(5);
// Error: // Uncaught Exception: Method Invocation node.add : at Line: 18 : in file: <unknown
file> : node .add ( 5 )

Target exception: java.lang.IllegalArgumentException

java.lang.IllegalArgumentException
 at org.jboss.dmr.ModelValue.addChild(ModelValue.java:120)
 at org.jboss.dmr.ModelNode.add(ModelNode.java:1007)
 at org.jboss.dmr.ModelNode.add(ModelNode.java:761)
        ...
```

You can, however, use the `setEmptyList()` method to change the node's type, and then use `add`:

```
bsh % node.setEmptyList();
bsh % node.add(5);
bsh % print(node.toString());
[5]
```

Properties

The third public class in the jboss-dmr library is `org.jboss.dmr.Property`. A `Property` is a `String => ModelNode` tuple.

```
bsh % Property prop = new Property("stuff", list);
bsh % print(prop.toString());
org.jboss.dmr.Property@79a5f739
bsh % print(prop.getName());
stuff
bsh % print(prop.getValue());
[
    5,
    10,
    "A string",
    undefined,
    undefined,
    30
]
```

The property can be passed to `ModelNode.set`:

```
bsh % node.set(prop);
bsh % print(node.getType());
PROPERTY
```

The text format for a node of `ModelType.PROPERTY` is:

```
bsh % print(node.toString());
("stuff" => [
    5,
    10,
    "A string",
    undefined,
    undefined,
    30
])
```

Directly instantiating a `Property` via its constructor is not common. More typically one of the two argument `ModelNode.add` or `ModelNode.set` variants is used. The first argument is the property name:

```
bsh % ModelNode simpleProp = new ModelNode();
bsh % simpleProp.set("enabled", true);
bsh % print(simpleProp.toString());
("enabled" => true)
bsh % print(simpleProp.getType());
PROPERTY
bsh % ModelNode propList = new ModelNode();
bsh % propList.add("min", 1);
bsh % propList.add("max", 10);
bsh % print(propList.toString());
[
    ("min" => 1),
    ("max" => 10)
]
bsh % print(propList.getType());
LIST
bsh % print(propList.get(0).getType());
PROPERTY
```

The `asPropertyList()` method provides easy access to a `List<Property>`:

```
bsh % for (Property prop : propList.asPropertyList()) {
print(prop.getName() + " = " + prop.getValue());
}
min = 1
max = 10
```

ModelType.OBJECT

The most powerful and most commonly used complex value type in jboss-dmr is `ModelType.OBJECT`. A `ModelNode` whose value is `ModelType.OBJECT` internally maintains a `Map<String, ModelNode`.

Use the `get` method variant that takes a string argument to add an entry to the map. If no entry exists under the given name, a new entry is added with a the value being a `ModelType.UNDEFINED` node. The node is returned:

```
bsh % ModelNode range = new ModelNode();
bsh % ModelNode min = range.get("min");
bsh % print(range.toString());
{"min" => undefined}
bsh % min.set(2);
bsh % print(range.toString());
{"min" => 2}
```

Again **it is important to remember that the `get` operation may mutate the state of a model node by adding a new entry.** *It is not a read-only operation.*

Since `get` will never return `null`, a common pattern is to use method chaining to create the key/value pair:

```
bsh % range.get("max").set(10);
bsh % print(range.toString());
{
    "min" => 2,
    "max" => 10
}
```

A call to `get` passing an already existing key will of course return the same model node as was returned the first time `get` was called with that key:

```
bsh % print(min == range.get("min"));
true
```

Multiple parameters can be passed to `get`. This is a simple way to traverse a tree made up of `ModelType.OBJECT` nodes. Again, `get` may mutate the node on which it is invoked; e.g. it will actually create the tree if nodes do not exist. This next example uses a workaround to get beanshell to handle the overloaded `get` method that takes a variable number of arguments:

```
bsh % String[] varargs = { "US", "Missouri", "St. Louis" };
bsh % salesTerritories.get(varargs).set("Brian");
bsh % print(salesTerritories.toString());
{"US" => {"Missouri" => {"St. Louis" => "Brian"}}}
```

The normal syntax would be:

```
salesTerritories.get("US", "Missouri", "St. Louis").set("Brian");
```

The key/value pairs in the map can be accessed as a `List<Property`:

```
bsh % for (Property prop : range.asPropertyList()) {
print(prop.getName() + " = " + prop.getValue());
}
min = 2
```

The semantics of the backing map in a node of `ModelType.OBJECT` are those of a `LinkedHashMap`. The map remembers the order in which key/value pairs are added. This is relevant when iterating over the pairs after calling `asPropertyList()` and for controlling the order in which key/value pairs appear in the output from `toString()`.

Since the `get` method will actually mutate the state of a node if the given key does not exist, `ModelNode` provides a couple methods to let you check whether the entry is there. The `has` method simply does that:

```
bsh % print(range.has("unit"));
false
bsh % print(range.has("min"));
true
```

Very often, the need is to not only know whether the key/value pair exists, but whether the value is defined (i.e. not `ModelType.UNDEFINED`. This kind of check is analogous to checking whether a field in a Java class has a null value. The `hasDefined` lets you do this:

```
bsh % print(range.hasDefined("unit"));
false
bsh % // Establish an undefined child 'unit';
bsh % range.get("unit");
bsh % print(range.toString());
{
    "min" => 2,
    "max" => 10,
    "unit" => undefined
}
bsh % print(range.hasDefined("unit"));
false
bsh % range.get("unit").set("meters");
bsh % print(range.hasDefined("unit"));
true
```

ModelType.EXPRESSION

A value of type `ModelType.EXPRESSION` is stored as a string, but can later be *resolved* to different value. The string has a special syntax that should be familiar to those who have used the system property substitution feature in previous JBoss AS releases.

```
[<prefix>][${<system-property-name>[:<default-value>]}][<suffix>]*
```

For example:

```
${queue.length}
http://${host}
http://${host:localhost}:${port:8080}/index.html
```

Use the `setExpression` method to set a node's value to type expression:

```
bsh % ModelNode expression = new ModelNode();
bsh % expression.setExpression("${queue.length}");
bsh % print(expression.getType());
EXPRESSION
```

Calling `asString()` returns the same string that was input:

```
bsh % print(expression.asString());
${queue.length}
```

However, calling `toString()` tells you that this node's value is not of `ModelType.STRING`:

```
bsh % print(expression.toString());
expression "${queue.length}"
```

When the `resolve` operation is called, the string is parsed and any embedded system properties are resolved against the JVM's current system property values. A new `ModelNode` is returned whose value is the resolved string:

```
bsh % System.setProperty("queue.length", "10");
bsh % ModelNode resolved = expression.resolve();
bsh % print(resolved.asInt());
10
```

Note that the type of the `ModelNode` returned by `resolve()` is `ModelType.STRING`:

```
bsh % print(resolved.getType());
STRING
```

The `resolved.asInt()` call in the previous example only worked because the string "10" happens to be convertible into the int 10.

Calling `resolve()` has no effect on the value of the node on which the method is invoked:

```
bsh % resolved = expression.resolve();
bsh % print(resolved.toString());
"10"
bsh % print(expression.toString());
expression "${queue.length}"
```

If an expression cannot be resolved, `resolve` just uses the original string. The string can include more than one system property substitution:

```
bsh % expression.setExpression("http://${host}:${port}/index.html");
bsh % resolved = expression.resolve();
bsh % print(resolved.asString());
http://${host}:${port}/index.html
```

The expression can optionally include a default value, separated from the name of the system property by a colon:

```
bsh % expression.setExpression("http://${host:localhost}:${port:8080}/index.html");
bsh % resolved = expression.resolve();
bsh % print(resolved.asString());
http://localhost:8080/index.html
```

Actually including a system property substitution in the expression is not required:

```
bsh % expression.setExpression("no system property");
bsh % resolved = expression.resolve();
bsh % print(resolved.asString());
no system property
bsh % print(expression.toString());
expression "no system property"
```

The `resolve` method works on nodes of other types as well; it returns a copy without attempting any real resolution:

```
bsh % ModelNode basic = new ModelNode();
bsh % basic.set(10);
bsh % resolved = basic.resolve();
bsh % print(resolved.getType());
INT
bsh % resolved.set(5);
bsh % print(resolved.asInt());
5
bsh % print(basic.asInt());
10
```

ModelType.TYPE

You can also pass one of the values of the `ModelType` enum to `set`:

```
bsh % ModelNode type = new ModelNode();
bsh % type.set(ModelType.LIST);
bsh % print(type.getType());
TYPE
bsh % print(type.toString());
LIST
```

This is useful when using a `ModelNode` data structure to describe another `ModelNode` data structure.

Full list of ModelNode types

BIG_DECIMAL
BIG_INTEGER
BOOLEAN
BYTES
DOUBLE
EXPRESSION
INT
LIST
LONG
OBJECT
PROPERTY
STRING
TYPE
UNDEFINED

Text representation of a ModelNode

TODO – document the grammar

JSON representation of a ModelNode

TODO – document the grammar

5.17.3 Description of the Management Model

A detailed description of the resources, attributes and operations that make up the management model provided by an individual WildFly instance or by any Domain Controller or slave Host Controller process can be queried using the `read-resource-description`, `read-operation-names`, `read-operation-description` and `read-child-types` operations described in the Global operations section. In this section we provide details on what's included in those descriptions.

Description of the WildFly Managed Resources

All portions of the management model exposed by WildFly are addressable via an ordered list of key/value pairs. For each addressable *Management Resource*, the following descriptive information will be available:

- `description` – String – text description of this portion of the model
- `min-occurs` – int, either 0 or 1 – Minimum number of resources of this type that must exist in a valid model. If not present, the default value is 0.
- `max-occurs` – int – Maximum number of resources of this type that may exist in a valid model. If not present, the default value depends upon the value of the final key/value pair in the address of the described resource. If this value is '*', the default value is Integer.MAX_VALUE, i.e. there is no limit. If this value is some other string, the default value is 1.
- `attributes` – Map of String (the attribute name) to complex structure – the configuration attributes available in this portion of the model. See below for the representation of each attribute.
- `operations` – Map of String (the operation name) to complex structure – the operations that can be targetted at this address. See below for the representation of each operation.
- `children` – Map of String (the type of child) to complex structure – the relationship of this portion of the model to other addressable portions of the model. See below for the representation of each child relationship.
- `head-comment-allowed` – boolean – indicates whether this portion of the model can store an XML comment that would be written in the persistent form of the model (e.g. domain.xml) before the start of the XML element that represents this portion of the model. This item is optional, and if not present defaults to true. (Note: storing XML comments in the in-memory model is not currently supported. This description key is for future use.)
- `tail-comment-allowed` – boolean – similar to `head-comment-allowed`, but indicates whether a comment just before the close of the XML element is supported. A tail comment can only be supported if the element has child elements, in which case a comment can be inserted between the final child element and the element's closing tag. This item is optional, and if not present defaults to true. (Note: storing XML comments in the in-memory model is not currently supported. This description key is for future use.)

For example:

```
{
     "description => "A manageable resource",
     "tail-comment-allowed" => false,
     "attributes" => {
          "foo" => {
               .... details of attribute foo
          }
      },
     "operations" => {
          "start" => {
               .... details of the start operation
          }
      },
     "children" => {
          "bar" => {
               .... details of the relationship with children of type "bar"
          }
      }
}
```

Description of an Attribute

An attribute is a portion of the management model that is not directly addressable. Instead, it is conceptually a property of an addressable management resource. For each attribute in the model, the following descriptive information will be available:

- `description` – String – text description of the attribute
- `type` – `org.jboss.dmr.ModelType` – the type of the attribute value. One of the enum values BIG_DECIMAL, BIG_INTEGER, BOOLEAN, BYTES, DOUBLE, INT, LIST, LONG, OBJECT, PROPERTY, STRING. Most of these are self-explanatory. An OBJECT will be represented in the detyped model as a map of string keys to values of some other legal type, conceptually similar to a `javax.management.openmbean.CompositeData`. A PROPERTY is a single key/value pair, where the key is a string, and the value is of some other legal type.
- `value-type` – ModelType or complex structure – Only present if type is LIST or OBJECT. If all elements in the LIST or all the values of the OBJECT type are of the same type, this will be one of the ModelType enums BIG_DECIMAL, BIG_INTEGER, BOOLEAN, BYTES, DOUBLE, INT, LONG, STRING. Otherwise, `value-type` will detail the structure of the attribute value, enumerating the value's fields and the type of their value. So, an attribute with a `type` of LIST and a `value-type` value of `ModelType.STRING` is analogous to a Java `List<String>`, while one with a `value-type` value of `ModelType.INT` is analogous to a Java `List<Integer>`. An attribute with a `type` of OBJECT and a `value-type` value of `ModelType.STRING` is analogous to a Java `Map<String, String>`. An attribute with a `type` of OBJECT and a `value-type` whose value is not of type `ModelType` represents a fully-defined complex object, with the object's legal fields and their values described.
- `expressions-allowed` – boolean – indicates whether the value of the attribute may be of type `ModelType.EXPRESSION`, instead of its standard type (see `type` and `value-type` above for discussion of an attribute's standard type.) A value of ModelType.EXPRESSION contains a system-property substitution expression that the server will resolve against the server-side system property map before using the value. For example, an attribute named max-threads may have an expression value of `${example.pool.max-threads:10}` instead of just 10. Default value if not present is false.
- `required` – boolean – true if the attribute must have a defined value in a representation of its portion of the model unless another attribute included in a list of `alternatives` is defined; false if it may be undefined (implying a null value) even in the absence of alternatives. If not present, true is the default.
- `nillable` – boolean – true if the attribute might not have a defined value in a representation of its portion of the model. A nillable attribute may
  be undefined either because it is not `required` or because it is required but has `alternatives` and one of the alternatives is defined.
- `storage` – String – Either "configuration" or "runtime". If "configuration", the attribute's value is stored as part of the persistent configuration (e.g. in domain.xml, host.xml or standalone.xml.) If "runtime" the attribute's value is not stored in the persistent configuration; the value only exists as long as the resource is running.

- `access-type` – String – One of "read-only", "read-write" or "metric". Whether an attribute value can be written, or can only read. A "metric" is a read-only attribute whose value is not stored in the persistent configuration, and whose value may change due to activity on the server. If an attribute is "read-write", the resource will expose an operation named "write-attribute" whose "name" parameter will accept this attribute's name and whose "value" parameter will accept a valid value for this attribute. That operation will be the standard means of updating this attribute's value.
- `restart-required` – String – One of "no-services", "all-services", "resource-services" or "jvm". Only relevant to attributes whose access-type is read-write. Indicates whether execution of a write-attribute operation whose name parameter specifies this attribute requires a restart of services (or an entire JVM) in order for the change to take effect in the runtime . See discussion of "Applying Updates to Runtime Services" below. Default value is "no-services".
- `default` – the default value for the attribute that will be used in runtime services if the attribute is not explicitly defined and no other attributes listed as `alternatives` are defined.
- `alternatives` – List of string – Indicates an exclusive relationship between attributes. If this attribute is defined, the other attributes listed in this descriptor's value should be undefined, even if their `required` descriptor says true; i.e. the presence of this attribute satisfies the requirement. Note that an attribute that is not explicitly configured but has a `default` value is still regarded as not being defined for purposes of checking whether the exclusive relationship has been violated. Default is undefined; i.e. this does not apply to most attributes.
- `requires` – List of string – Indicates that if this attribute has a value (other than undefined), the other attributes listed in this descriptor's value must also have a value, even if their required descriptor says false. This would typically be used in conjunction with alternatives. For example, attributes "a" and "b" are required, but are alternatives to each other; "c" and "d" are optional. But "b" requires "c" and "d", so if "b" is used, "c" and "d" must also be defined. Default is undefined; i.e. this does not apply to most attributes.
- `capability-reference` – string – if defined indicates that this attribute's value specifies the dynamic portion of the name of the specified capability provided by another resource. This indicates the attribute is a reference to another area of the management model. (Note that at present some attributes that reference other areas of the model may not provide this information.)
- `head-comment-allowed` – boolean – indicates whether the model can store an XML comment that would be written in the persistent form of the model (e.g. domain.xml) before the start of the XML element that represents this attribute. This item is optional, and if not present defaults to false. (This is a different default from what is used for an entire management resource, since model attributes often map to XML attributes, which don't allow comments.) (Note: storing XML comments in the in-memory model is not currently supported. This description key is for future use.)
- `tail-comment-allowed` – boolean – similar to head-comment-allowed, but indicates whether a comment just before the close of the XML element is supported. A tail comment can only be supported if the element has child elements, in which case a comment can be inserted between the final child element and the element's closing tag. This item is optional, and if not present defaults to false. (This is a different default from what is used for an entire management resource, since model attributes often map to XML attributes, which don't allow comments.) (Note: storing XML comments in the in-memory model is not currently supported. This description key is for future use.)
- arbitrary key/value pairs that further describe the attribute value, e.g. "max" => 2. See "Arbitrary Descriptors" below.

Some examples:

```
"foo" => {
     "description" => "The foo",
     "type" => INT,
     "max" => 2
}
```

```
"bar" => {
     "description" => "The bar",
     "type" => OBJECT,
     "value-type" => {
          "size" => INT,
          "color" => STRING
     }
}
```

Description of an Operation

A management resource may have operations associated with it. The description of an operation will include the following information:

- `operation-name` – String – the name of the operation
- `description` – String – text description of the operation
- `request-properties` – Map of String to complex structure – description of the parameters of the operation. Keys are the names of the parameters, values are descriptions of the parameter value types. See below for details on the description of parameter value types.
- `reply-properties` – complex structure, or empty – description of the return value of the operation, with an empty node meaning void. See below for details on the description of operation return value types.
- `restart-required` – String – One of "no-services", "all-services", "resource-services" or "jvm". Indicates whether the operation makes a configuration change that requires a restart of services (or an entire JVM) in order for the change to take effect in the runtime. See discussion of "Applying Updates to Runtime Services" below. Default value is "no-services".

Description of an Operation Parameter or Return Value

- `description` – String – text description of the parameter or return value
- `type` – `org.jboss.dmr.ModelType` – the type of the parameter or return value. One of the enum values BIG_DECIMAL, BIG_INTEGER, BOOLEAN, BYTES, DOUBLE, INT, LIST, LONG, OBJECT, PROPERTY, STRING.

- `value-type` – `ModelType` or complex structure – Only present if type is LIST or OBJECT. If all elements in the LIST or all the values of the OBJECT type are of the same type, this will be one of the ModelType enums BIG_DECIMAL, BIG_INTEGER, BOOLEAN, BYTES, DOUBLE, INT, LIST, LONG, PROPERTY, STRING. Otherwise, value-type will detail the structure of the attribute value, enumerating the value's fields and the type of their value.So, a parameter with a `type` of LIST and a `value-type` value of `ModelType.STRING` is analogous to a Java `List<String>`, while one with a `value-type` value of `ModelType.INT` is analogous to a Java `List<Integer>`. A parameter with a `type` of OBJECT and a `value-type` value of `ModelType.STRING` is analogous to a Java `Map<String, String>`. A parameter with a `type` of OBJECT and a `value-type` whose value is not of type `ModelType` represents a fully-defined complex object, with the object's legal fields and their values described.
- `expressions-allowed` – boolean – indicates whether the value of the the parameter or return value may be of type `ModelType.EXPRESSION`, instead its standard type (see type and value-type above for discussion of the standard type.) A value of ModelType.EXPRESSION contains a system-property substitution expression that the server will resolve against the server-side system property map before using the value. For example, a parameter named max-threads may have an expression value of `${example.pool.max-threads:10}` instead of just 10. Default value if not present is false.
- `required` – boolean – true if the parameter or return value must have a defined value in the operation or response unless another item included in a list of `alternatives` is defined; false if it may be undefined (implying a null value) even in the absence of alternatives. If not present, true is the default.
- `nillable` – boolean – true if the parameter or return value might not have a defined value in a representation of its portion of the model. A nillable parameter or return value may be undefined either because it is not `required` or because it is required but has `alternatives` and one of the alternatives is defined.
- `default` – the default value for the parameter that will be used in runtime services if the parameter is not explicitly defined and no other parameters listed as `alternatives` are defined.
- `restart-required` – String – One of "no-services", "all-services", "resource-services" or "jvm". Only relevant to attributes whose access-type is read-write. Indicates whether execution of a write-attribute operation whose name parameter specifies this attribute requires a restart of services (or an entire JVM) in order for the change to take effect in the runtime . See discussion of "Applying Updates to Runtime Services" below. Default value is "no-services".
- `alternatives` – List of string – Indicates an exclusive relationship between parameters. If this attribute is defined, the other parameters listed in this descriptor's value should be undefined, even if their required descriptor says true; i.e. the presence of this parameter satisfies the requirement. Note that an parameer that is not explicitly configured but has a `default` value is still regarded as not being defined for purposes of checking whether the exclusive relationship has been violated. Default is undefined; i.e. this does not apply to most parameters.
- `requires` – List of string – Indicates that if this parameter has a value (other than undefined), the other parameters listed in this descriptor's value must also have a value, even if their required descriptor says false. This would typically be used in conjunction with alternatives. For example, parameters "a" and "b" are required, but are alternatives to each other; "c" and "d" are optional. But "b" requires "c" and "d", so if "b" is used, "c" and "d" must also be defined. Default is undefined; i.e. this does not apply to most parameters.

- arbitrary key/value pairs that further describe the attribute value, e.g. "max" =>2. See "Arbitrary Descriptors" below.

Arbitrary Descriptors

The description of an attribute, operation parameter or operation return value type can include arbitrary key/value pairs that provide extra information. Whether a particular key/value pair is present depends on the context, e.g. a pair with key "max" would probably only occur as part of the description of some numeric type.

Following are standard keys and their expected value type. If descriptor authors want to add an arbitrary key/value pair to some descriptor and the semantic matches the meaning of one of the following items, the standard key/value type must be used.

- `min` – int – the minimum value of some numeric type. The absence of this item implies there is no minimum value.
- `max` – int – the maximum value of some numeric type. The absence of this item implies there is no maximum value.
- `min-length` – int – the minimum length of some string, list or byte[] type. The absence of this item implies a minimum length of zero.
- `max-length` – int – the maximum length of some string, list or byte[]. The absence of this item implies there is no maximum value.
- `allowed` – List – a list of legal values. The type of the elements in the list should match the type of the attribute.
- `unit` - The unit of the value, if one is applicable - e.g. ns, ms, s, m, h, KB, MB, TB. See the `org.jboss.as.controller.client.helpers.MeasurementUnit` in the org.jboss.as:jboss-as-controller-client artifact for a listing of legal measurement units..

Some examples:

```
{
     "operation-name" => "incrementFoo",
     "description" => "Increase the value of the 'foo' attribute by the given amount",
     "request-properties" => {
          "increment" => {
               "type" => INT,
               "description" => "The amount to increment",
               "required" => true
     }},
     "reply-properties" => {
               "type" => INT,
               "description" => "The new value",
     }
}
```

```
{
     "operation-name" => "start",
     "description" => "Starts the thing",
     "request-properties" => {},
     "reply-properties" => {}
}
```

Description of Parent/Child Relationships

The address used to target an addressable portion of the model must be an ordered list of key value pairs. The effect of this requirement is the addressable portions of the model naturally form a tree structure, with parent nodes in the tree defining what the valid keys are and the children defining what the valid values are. The parent node also defines the cardinality of the relationship. The description of the parent node includes a children element that describes these relationships:

```
{
     ....
     "children" => {
          "connector" => {
               .... description of the relationship with children of type "connector"
          },
          "virtual-host" => {
               .... description of the relationship with children of type "virtual-host"
          }
}
```

The description of each relationship will include the following elements:

- `description` – String – text description of the relationship
- `model-description` – either "undefined" or a complex structure – This is a node of ModelType.OBJECT, the keys of which are legal values for the value portion of the address of a resource of this type, with the special character '*' indicating the value portion can have an arbitrary value. The values in the node are the full description of the particular child resource (its text description, attributes, operations, children) as detailed above. This `model-description` may also be "undefined", i.e. a null value, if the query that asked for the parent node's description did not include the "recursive" param set to true.

Example with if the recursive flag was set to true:

```
{
     "description" => "The connectors used to handle client connections",
     "model-description" => {
          "*" => {
              "description" => "Handles client connections",
              "min-occurs" => 1,
              "attributes => {
                   ... details of children as documented above
              },
              "operations" => {
                   .... details of operations as documented above
              },
              "children" => {
                   .... details of the children's children
              }
          }
     }
}
```

If the recursive flag was false:

```
{
     "description" => "The connectors used to handle client connections",
     "model-description" => undefined
}
```

Applying Updates to Runtime Services

An attribute or operation description may include a "`restart-required`" descriptor; this section is an explanation of the meaning of that descriptor.

An operation that changes a management resource's persistent configuration usually can also also affect a runtime service associated with the resource. For example, there is a runtime service associated with any host.xml or standalone.xml `<interface>` element; other services in the runtime depend on that service to provide the `InetAddress` associated with the interface. In many cases, an update to a resource's persistent configuration can be immediately applied to the associated runtime service. The runtime service's state is updated to reflect the new value(s).

However, in many cases the runtime service's state cannot be updated without restarting the service. Restarting a service can have broad effects. A restart of a service A will trigger a restart of other services B, C and D that depend A, triggering a restart of services that depend on B, C and D, etc. Those service restarts may very well disrupt handling of end-user requests.

Because restarting a service can be disruptive to end-user request handling, the handlers for management operations will not restart any service without some form of explicit instruction from the end user indicating a service restart is desired. In a few cases, simply executing the operation is an indication the user wants services to restart (e.g. a `/host=master/server-config=server-one:restart` operation in a managed domain, or a `/:reload` operation on a standalone server.) For all other cases, if an operation (or attribute write) cannot be performed without restarting a service, the metadata describing the operation or attribute will include a "`restart-required`" descriptor whose value indicates what is necessary for the operation to affect the runtime:

- `no-services` – Applying the operation to the runtime does not require the restart of any services. This value is the default if the restart-required descriptor is not present.
- `all-services` – The operation can only immediately update the persistent configuration; applying the operation to the runtime will require a subsequent restart of all services in the affected VM. Executing the operation will put the server into a "`reload-required`" state. Until a restart of all services is performed the response to this operation and to any subsequent operation will include a response header "`process-state" => "reload-required`". For a standalone server, a restart of all services can be accomplished by executing the `/:reload` CLI command. For a server in a managed domain, restarting all services currently requires a full restart of the affected server VM (e.g. `/host=master/server-config=server-one:restart`).
- `jvm` --The operation can only immediately update the persistent configuration; applying the operation to the runtime will require a full process restart (i.e. stop the JVM and launch a new JVM). Executing the operation will put the server into a "`restart-required`" state. Until a restart is performed the response to this operation and to any subsequent operation will include a response header "`process-state" => "restart-required`". For a standalone server, a full process restart requires first stopping the server via OS-level operations (Ctrl-C, kill) or via the `/:shutdown` CLI command, and then starting the server again from the command line. For a server in a managed domain, restarting a server requires executing the `/host=<host>/server-config=<server>:restart` operation.
- `resource-services` – The operation can only immediately update the persistent configuration; applying the operation to the runtime will require a subsequent restart of some services associated with the resource. If the operation includes the request header `"allow-resource-service-restart" => true`, the handler for the operation will go ahead and restart the runtime service. Otherwise executing the operation will put the server into a "`reload-required`" state. (See the discussion of "`all-services`" above for more on the "`reload-required`" state.)

5.17.4 The native management API

A standalone WildFly process, or a managed domain Domain Controller or slave Host Controller process can be configured to listen for remote management requests using its "native management interface":

```
<native-interface interface="management" port="9999" security-realm="ManagementRealm"/>
```

(See standalone/configuration/standalone.xml or domain/configuration/host.xml)

The CLI tool that comes with the application server uses this interface, and user can develop custom clients that use it as well. In this section we'll cover the basics on how to develop such a client. We'll also cover details on the format of low-level management operation requests and responses – information that should prove useful for users of the CLI tool as well.

Native Management Client Dependencies

The native management interface uses an open protocol based on the JBoss Remoting library. JBoss Remoting is used to establish a communication channel from the client to the process being managed. Once the communication channel is established the primary traffic over the channel is management requests initiated by the client and asynchronous responses from the target process.

A custom Java-based client should have the maven artifact `org.jboss.as:jboss-as-controller-client` and its dependencies on the classpath. The other dependencies are:

Maven Artifact	Purpose
`org.jboss.remoting:jboss-remoting`	Remote communication
`org.jboss:jboss-dmr`	Detyped representation of the management model
`org.jboss.as:jboss-as-protocol`	Wire protocol for remote WildFly management
`org.jboss.sasl:jboss-sasl`	SASL authentication
`org.jboss.xnio:xnio-api`	Non-blocking IO
`org.jboss.xnio:xnio-nio`	Non-blocking IO
`org.jboss.logging:jboss-logging`	Logging
`org.jboss.threads:jboss-threads`	Thread management
`org.jboss.marshalling:jboss-marshalling`	Marshalling and unmarshalling data to/from streams

The client API is entirely within the `org.jboss.as:jboss-as-controller-client` artifact; the other dependencies are part of the internal implementation of `org.jboss.as:jboss-as-controller-client` and are not compile-time dependencies of any custom client based on it.

The management protocol is an open protocol, so a completely custom client could be developed without using these libraries (e.g. using Python or some other language.)

Working with a ModelControllerClient

The `org.jboss.as.controller.client.ModelControllerClient` class is the main class a custom client would use to manage a WildFly server instance or a Domain Controller or slave Host Controller.

The custom client must have maven artifact `org.jboss.as:jboss-as-controller-client` and its dependencies on the classpath.

Creating the ModelControllerClient

To create a management client that can connect to your target process's native management socket, simply:

```
ModelControllerClient client =
ModelControllerClient.Factory.create(InetAddress.getByName("localhost"), 9999);
```

The address and port are what is configured in the target process' `<management><management-interfaces><native-interface.../>` element.

Typically, however, the native management interface will be secured, requiring clients to authenticate. On the client side, the custom client will need to provide the user's authentication credentials, obtained in whatever manner is appropriate for the client (e.g. from a dialog box in a GUI-based client.) Access to these credentials is provided by passing in an implementation of the `javax.security.auth.callback.CallbackHandler` interface. For example:

```
static ModelControllerClient createClient(final InetAddress host, final int port,
                  final String username, final char[] password, final String securityRealmName)
{

    final CallbackHandler callbackHandler = new CallbackHandler() {

        public void handle(Callback[] callbacks) throws IOException,
UnsupportedCallbackException {
            for (Callback current : callbacks) {
                if (current instanceof NameCallback) {
                    NameCallback ncb = (NameCallback) current;
                    ncb.setName(username);
                } else if (current instanceof PasswordCallback) {
                    PasswordCallback pcb = (PasswordCallback) current;
                    pcb.setPassword(password.toCharArray());
                } else if (current instanceof RealmCallback) {
                    RealmCallback rcb = (RealmCallback) current;
                    rcb.setText(rcb.getDefaultText());
                } else {
                    throw new UnsupportedCallbackException(current);
                }
            }
        }
    };

    return ModelControllerClient.Factory.create(host, port, callbackHandler);
}
```

Creating an operation request object

Management requests are formulated using the `org.jboss.dmr.ModelNode` class from the `jboss-dmr` library. The `jboss-dmr` library allows the complete WildFly management model to be expressed using a very small number of Java types. See Detyped management and the jboss-dmr library for full details on using this library.

Let's show an example of creating an operation request object that can be used to read the resource description for the web subsystem's HTTP connector:

```
ModelNode op = new ModelNode();
op.get("operation").set("read-resource-description");

ModelNode address = op.get("address");
address.add("subsystem", "web");
address.add("connector", "http");

op.get("recursive").set(true);
op.get("operations").set(true);
```

What we've done here is created a ModelNode of type `ModelType.OBJECT` with the following fields:

- `operation` – the name of the operation to invoke. All operation requests **must** include this field and its value must be a String.
- `address` – the address of the resource to invoke the operation against. This field's must be of `ModelType.LIST` with each element in the list being a `ModelType.PROPERTY`. If this field is omitted the operation will target the root resource. The operation can be targeted at any address in the management model; here we are targeting it at the resource for the web subsystem's http connector.

In this case, the request includes two optional parameters:

- `recursive` – true means you want the description of child resources under this resource. Default is false
- `operations` – true means you want the description of operations exposed by the resource to be included. Default is false.

Different operations take different parameters, and some take no parameters at all.

See Format of a Detyped Operation Request for full details on the structure of a ModelNode that will represent an operation request.

The example above produces an operation request ModelNode equivalent to what the CLI produces internally when it parses and executes the following low-level CLI command:

```
[localhost:9999 /]
/subsystem=web/connector=http:read-resource-description(recursive=true,operations=true)
```

Execute the operation and manipulate the result:

The `execute` method sends the operation request ModelNode to the process being managed and returns a ModelNode the contains the process' response:

```
ModelNode returnVal = client.execute(op);
System.out.println(returnVal.get("result").toString());
```

See Format of a Detyped Operation Response for general details on the structure of the "returnVal" ModelNode.

The `execute` operation shown above will block the calling thread until the response is received from the process being managed. `ModelControllerClient` also exposes and API allowing asynchronous invocation:

```
Future<ModelNode> future = client.executeAsync(op);
. . .  // do other stuff
ModelNode returnVal = future.get();
System.out.println(returnVal.get("result").toString());
```

Close the ModelControllerClient

A `ModelControllerClient` can be reused for multiple requests. Creating a new `ModelControllerClient` for each request is an anti-pattern. However, when the `ModelControllerClient` is no longer needed, it should always be explicitly closed, allowing it to close down any connections to the process it was managing and release other resources:

```
client.close();
```

Format of a Detyped Operation Request

The basic method a user of the WildFly 8 programmatic management API would use is very simple:

```
ModelNode execute(ModelNode operation) throws IOException;
```

where the return value is the detyped representation of the response, and `operation` is the detyped representation of the operation being invoked.

The purpose of this section is to document the structure of `operation`.

See Format of a Detyped Operation Response for a discussion of the format of the response.

Simple Operations

A text representation of simple operation would look like this:

```
{
    "operation" => "write-attribute",
    "address" => [
        ("profile" => "production"),
        ("subsystem" => "threads"),
        ("bounded-queue-thread-pool" => "pool1")
    ],
    "name" => "count",
    "value" => 20
}
```

Java code to produce that output would be:

```
ModelNode op = new ModelNode();
op.get("operation").set("write-attribute");
ModelNode addr = op.get("address");
addr.add("profile", "production");
addr.add("subsystem", "threads");
addr.add("bounded-queue-thread-pool", "pool1");
op.get("name").set("count");
op.get("value").set(20);
System.out.println(op);
```

The order in which the outermost elements appear in the request is not relevant. The required elements are:

- `operation` – String – The name of the operation being invoked.
- `address` – the address of the managed resource against which the request should be executed. If not set, the address is the root resource. The address is an ordered list of key-value pairs describing where the resource resides in the overall management resource tree. Management resources are organized in a tree, so the order in which elements in the address occur is important.

The other key/value pairs are parameter names and their values. The names and values should match what is specified in the operation's description.

Parameters may have any name, except for the reserved words `operation`, `address` and `operation-headers`.

Operation Headers

Besides the special operation and address values discussed above, operation requests can also include special "header" values that help control how the operation executes. These headers are created under the special reserved word `operation-headers`:

```
ModelNode op = new ModelNode();
op.get("operation").set("write-attribute");
ModelNode addr = op.get("address");
addr.add("base", "domain");
addr.add("profile", "production");
addr.add("subsystem", "threads");
addr.add("bounded-queue-thread-pool", "pool1");
op.get("name").set("count");
op.get("value").set(20);
op.get("operation-headers", "rollback-on-runtime-failure").set(false);
System.out.println(op);
```

This produces:

```
{
    "operation" => "write-attribute",
    "address" => [
        ("profile" => "production"),
        ("subsystem" => "threads"),
        ("bounded-queue-thread-pool" => "pool1")
    ],
    "name" => "count",
    "value" => 20,
    "operation-headers" => {
        "rollback-on-runtime-failure => false
    }
}
```

The following operation headers are supported:

- `rollback-on-runtime-failure` – boolean, optional, defaults to true. Whether an operation that successfully updates the persistent configuration model should be reverted if it fails to apply to the runtime. Operations that affect the persistent configuration are applied in two stages – first to the configuration model and then to the actual running services. If there is an error applying to the configuration model the operation will be aborted with no configuration change and no change to running services will be attempted. However, operations are allowed to change the configuration model even if there is a failure to apply the change to the running services – if and only if this `rollback-on-runtime-failure` header is set to `false`. So, this header only deals with what happens if there is a problem applying an operation to the running state of a server (e.g. actually increasing the size of a runtime thread pool.)
- `rollout-plan` – only relevant to requests made to a Domain Controller or Host Controller. See "Operations with a Rollout Plan" for details.
- `allow-resource-service-restart` – boolean, optional, defaults to false. Whether an operation that requires restarting some runtime services in order to take effect should do so. See discussion of `resource-services` in the "Applying Updates to Runtime Services" section of the Description of the Management Model section for further details.
- `roles` – String or list of strings. Name(s) of RBAC role(s) the permissions for which should be used when making access control decisions instead of those from the roles normally associated with the user invoking the operation. Only respected if the user is normally associated with a role with all permissions (i.e. SuperUser), meaning this can only be used to reduce permissions for a caller, not to increase permissions.
- `blocking-timeout` – int, optional, defaults to 300. Maximum time, in seconds, that the operation should block at various points waiting for completion. If this period is exceeded, the operation will roll back. Does not represent an overall maximum execution time for an operation; rather it is meant to serve as a sort of fail-safe measure to prevent problematic operations indefinitely tying up resources.

Composite Operations

The root resource for a Domain or Host Controller or an individual server will expose an operation named "`composite`". This operation executes a list of other operations as an atomic unit (although the atomicity requirement can be relaxed. The structure of the request for the "`composite`" operation has the same fundamental structure as a simple operation (i.e. operation name, address, params as key value pairs).

```
{
    "operation" => "composite",
    "address" => [],
    "steps" => [
         {
              "operation" => "write-attribute",
              "address" => [
                   ("profile" => "production"),
                   ("subsystem" => "threads"),
                   ("bounded-queue-thread-pool" => "pool1")
              ],
              "count" => "count",
              "value" => 20
         },
         {
              "operation" => "write-attribute",
              "address" => [
                   ("profile" => "production"),
                   ("subsystem" => "threads"),
                   ("bounded-queue-thread-pool" => "pool2")
              ],
              "name" => "count",
              "value" => 10
         }
    ],
    "operation-headers" => {
        "rollback-on-runtime-failure => false
    }
}
```

The "composite" operation takes a single parameter:

- `steps` – a list, where each item in the list has the same structure as a simple operation request. In the example above each of the two steps is modifying the thread pool configuration for a different pool. There need not be any particular relationship between the steps. Note that the `rollback-on-runtime-failure` and `rollout-plan` operation headers are not supported for the individual steps in a composite operation.

The `rollback-on-runtime-failure` operation header discussed above has a particular meaning when applied to a composite operation, controlling whether steps that successfully execute should be reverted if other steps fail at runtime. Note that if any steps modify the persistent configuration, and any of those steps fail, all steps will be reverted. Partial/incomplete changes to the persistent configuration are not allowed.

Operations with a Rollout Plan

Operations targeted at domain or host level resources can potentially impact multiple servers. Such operations can include a "rollout plan" detailing the sequence in which the operation should be applied to servers as well as policies for detailing whether the operation should be reverted if it fails to execute successfully on some servers.

If the operation includes a rollout plan, the structure is as follows:

```
{
    "operation" => "write-attribute",
    "address" => [
        ("profile" => "production"),
        ("subsystem" => "threads"),
        ("bounded-queue-thread-pool" => "pool1")
    ],
    "name" => "count",
    "value" => 20,
    "operation-headers" => {
        "rollout-plan" => {
            "in-series" => [
                {
                    "concurrent-groups" => {
                        "groupA" => {
                            "rolling-to-servers" => true,
                            "max-failure-percentage" => 20
                        },
                        "groupB" => undefined
                    }
                },
                {
                   "server-group" => {
                        "groupC" => {
                            "rolling-to-servers" => false,
                            "max-failed-servers" => 1
                        }
                    }
                },
                {
                    "concurrent-groups" => {
                        "groupD" => {
                            "rolling-to-servers" => true,
                            "max-failure-percentage" => 20
                        },
                        "groupE" => undefined
                    }
                }
            ],
            "rollback-across-groups" => true
        }
    }
}
```

As you can see, the rollout plan is another structure in the operation-headers section. The root node of the structure allows two children:

- `in-series` – a list – A list of activities that are to be performed in series, with each activity reaching completion before the next step is executed. Each activity involves the application of the operation to the servers in one or more server groups. See below for details on each element in the list.
- `rollback-across-groups` – boolean – indicates whether the need to rollback the operation on all the servers in one server group should trigger a rollback across all the server groups. This is an optional setting, and defaults to `false`.

Each element in the list under the `in-series` node must have one or the other of the following structures:

- `concurrent-groups` – a map of server group names to policies controlling how the operation should be applied to that server group. For each server group in the map, the operation may be applied concurrently. See below for details on the per-server-group policy configuration.
- `server-group` – a single key/value mapping of a server group name to a policy controlling how the operation should be applied to that server group. See below for details on the policy configuration. (Note: there is no difference in plan execution between this and a "`concurrent-groups`" map with a single entry.)

The policy controlling how the operation is applied to the servers within a server group has the following elements, each of which is optional:

- `rolling-to-servers` – boolean – If true, the operation will be applied to each server in the group in series. If false or not specified, the operation will be applied to the servers in the group concurrently.
- `max-failed-servers` – int – Maximum number of servers in the group that can fail to apply the operation before it should be reverted on all servers in the group. The default value if not specified is zero; i.e. failure on any server triggers rollback across the group.
- `max-failure-percentage` – int between 0 and 100 – Maximum percentage of the total number of servers in the group that can fail to apply the operation before it should be reverted on all servers in the group. The default value if not specified is zero; i.e. failure on any server triggers rollback across the group.

If both `max-failed-servers` and `max-failure-percentage` are set, `max-failure-percentage` takes precedence.

Looking at the (contrived) example above, application of the operation to the servers in the domain would be done in 3 phases. If the policy for any server group triggers a rollback of the operation across the server group, all other server groups will be rolled back as well. The 3 phases are:

1. Server groups groupA and groupB will have the operation applied concurrently. The operation will be applied to the servers in groupA in series, while all servers in groupB will handle the operation concurrently. If more than 20% of the servers in groupA fail to apply the operation, it will be rolled back across that group. If any servers in groupB fail to apply the operation it will be rolled back across that group.
2. Once all servers in groupA and groupB are complete, the operation will be applied to the servers in groupC. Those servers will handle the operation concurrently. If more than one server in groupC fails to apply the operation it will be rolled back across that group.
3. Once all servers in groupC are complete, server groups groupD and groupE will have the operation applied concurrently. The operation will be applied to the servers in groupD in series, while all servers in groupE will handle the operation concurrently. If more than 20% of the servers in groupD fail to apply the operation, it will be rolled back across that group. If any servers in groupE fail to apply the operation it will be rolled back across that group.

Default Rollout Plan

All operations that impact multiple servers will be executed with a rollout plan. However, actually specifying the rollout plan in the operation request is not required. If no `rollout-plan` operation header is specified, a default plan will be generated. The plan will have the following characteristics:

- There will only be a single high level phase. All server groups affected by the operation will have the operation applied concurrently.
- Within each server group, the operation will be applied to all servers concurrently.
- Failure on any server in a server group will cause rollback across the group.
- Failure of any server group will result in rollback of all other server groups.

Creating and reusing a Rollout Plan

Since a rollout plan may be quite complex, having to pass it as a header every time can become quickly painful. So instead we can store it in the model and then reference it when we want to use it.
To create a rollout plan you can use the operation `rollout-plan add` like this :

```
rollout-plan add --name=simple --content={"rollout-plan" => {"in-series" => [{"server-group" =>
{"main-server-group" => {"rolling-to-servers" => false,"max-failed-servers" => 1}}},
{"server-group" => {"other-server-group" => {"rolling-to-servers" =>
true,"max-failure-percentage" => 20}}}],"rollback-across-groups" => true}}
```

This will create a rollout plan called `simple` in the content repository.

```
[domain@192.168.1.20:9999 /]
/management-client-content=rollout-plans/rollout-plan=simple:read-resource
{
    "outcome" => "success",
    "result" => {
        "content" => {"rollout-plan" => {
            "in-series" => [
                {"server-group" => {"main-server-group" => {
                    "rolling-to-servers" => false,
                    "max-failed-servers" => 1
                }}},
                {"server-group" => {"other-server-group" => {
                    "rolling-to-servers" => true,
                    "max-failure-percentage" => 20
                }}}
            ],
            "rollback-across-groups" => true
        }},
        "hash" => bytes {
            0x13, 0x12, 0x76, 0x65, 0x8a, 0x28, 0xb8, 0xbc,
            0x34, 0x3c, 0xe9, 0xe6, 0x9f, 0x24, 0x05, 0xd2,
            0x30, 0xff, 0xa4, 0x34
        }
    }
}
```

Now you may reference the roolout plan in your command by adding a header just like this :

```
deploy /quickstart/ejb-in-war/target/wildfly-ejb-in-war.war --all-server-groups
--headers={rollout name=simple}
```

Format of a Detyped Operation Response

As noted previously, the basic method a user of the WildFly 8 programmatic management API would use is very simple:

```
ModelNode execute(ModelNode operation) throws IOException;
```

where the return value is the detyped representation of the response, and `operation` is the detyped representation of the operating being invoked.

The purpose of this section is to document the structure of the return value.

For the format of the request, see Format of a Detyped Operation Request.

Simple Responses

Simple responses are provided by the following types of operations:

- Non-composite operations that target a single server. (See below for more on composite operations).
- Non-composite operations that target a Domain Controller or slave Host Controller and don't require the responder to apply the operation on multiple servers and aggregate their results (e.g. a simple read of a domain configuration property.)

The response will always include a simple boolean outcome field, with one of three possible values:

- `success` – the operation executed successfully
- `failed` – the operation failed
- `cancelled` – the execution of the operation was cancelled. (This would be an unusual outcome for a simple operation which would generally very rapidly reach a point in its execution where it couldn't be cancelled.)

The other fields in the response will depend on whether the operation was successful.

The response for a failed operation:

```
{
    "outcome" => "failed",
    "failure-description" => "[JBAS-12345] Some failure message"
}
```

A response for a successful operation will include an additional field:

- `result` – the return value, or `undefined` for void operations or those that return null

A non-void result:

```
{
    "outcome" => "success",
    "result" => {
        "name" => "Brian",
        "age" => 22
    }
}
```

A void result:

```
{
    "outcome" => "success",
    "result" => undefined
}
```

The response for a cancelled operation has no other fields:

```
{
    "outcome" => "cancelled"
}
```

Response Headers

Besides the standard `outcome`, `result` and `failure-description` fields described above, the response may also include various headers that provide more information about the affect of the operation or about the overall state of the server. The headers will be child element under a field named `response-headers`. For example:

```
{
    "outcome" => "success",
    "result" => undefined,
    "response-headers" => {
        "operation-requires-reload" => true,
        "process-state" => "reload-required"
    }
}
```

A response header is typically related to whether an operation could be applied to the targeted runtime without requiring a restart of some or all services, or even of the target process itself. Please see the "Applying Updates to Runtime Services" section of the Description of the Management Model section for a discussion of the basic concepts related to what happens if an operation requires a service restart to be applied.

The current possible response headers are:

- `operation-requires-reload` – boolean – indicates that the specific operation that has generated this response requires a restart of all services in the process in order to take effect in the runtime. This would typically only have a value of 'true'; the absence of the header is the same as a value of 'false.'
- `operation-requires-restart` – boolean – indicates that the specific operation that has generated this response requires a full process restart in order to take effect in the runtime. This would typically only have a value of 'true'; the absence of the header is the same as a value of 'false.'
- `process-state` – enumeration – Provides information about the overall state of the target process. One of the following values:
  - `starting` – the process is starting
  - `running` – the process is in a normal running state. The `process-state` header would typically not be seen with this value; the absence of the header is the same as a value of 'running'.
  - `reload-required` – some operation (not necessarily this one) has executed that requires a restart of all services in order for a configuration change to take effect in the runtime.
  - `restart-required` – some operation (not necessarily this one) has executed that requires a full process restart in order for a configuration change to take effect in the runtime.
  - `stopping` – the process is stopping

Basic Composite Operation Responses

A composite operation is one that incorporates more than one simple operation in a list and executes them atomically. See the "Composite Operations" section for more information.

Basic composite responses are provided by the following types of operations:

- Composite operations that target a single server.
- Composite operations that target a Domain Controller or a slave Host Controller and don't require the responder to apply the operation on multiple servers and aggregate their results (e.g. a list of simple reads of domain configuration properties.)

The high level format of a basic composite operation response is largely the same as that of a simple operation response, although there is an important semantic difference. For a composite operation, the meaning of the outcome flag is controlled by the value of the operation request's `rollback-on-runtime-failure` header field. If that field was `false` (default is true), the outcome flag will be success if all steps were successfully applied to the persistent configuration even if **none** of the composite operation's steps was successfully applied to the runtime.

What's distinctive about a composite operation response is the `result` field. First, even if the operation was not successful, the `result` field will usually be present. (It won't be present if there was some sort of immediate failure that prevented the responder from even attempting to execute the individual operations.) Second, the content of the `result` field will be a map. Each entry in the map will record the result of an element in the `steps` parameter of the composite operation request. The key for each item in the map will be the string "`step-X`" where "X" is the 1-based index of the step's position in the request's `steps` list. So each individual operation in the composite operation will have its result recorded.

The individual operation results will have the same basic format as the simple operation results described above. However, there are some differences from the simple operation case when the individual operation's `outcome` flag is `failed`. These relate to the fact that in a composite operation, individual operations can be rolled back or not even attempted.

If an individual operation was not even attempted (because the overall operation was cancelled or, more likely, a prior operation failed):

```
{
    "outcome" => "cancelled"
}
```

An individual operation that failed and was rolled back:

```
{
    "outcome" => "failed",
    "failure-description" => "[JBAS-12345] Some failure message",
    "rolled-back" => true
}
```

An individual operation that itself succeeded but was rolled back due to failure of another operation:

```
{
    "outcome" => "failed",
    "result" => {
        "name" => "Brian",
        "age" => 22
    },
    "rolled-back" => true
}
```

An operation that failed and was rolled back:

```
{
    "outcome" => "failed",
    "failure-description" => "[JBAS-12345] Some failure message",
    "rolled-back" => true
}
```

Here's an example of the response for a successful 2 step composite operation:

```
{
    "outcome" => "success",
    "result" => [
        {
            "outcome" => "success",
            "result" => {
                "name" => "Brian",
                "age" => 22
            }
        },
        {
            "outcome" => "success",
            "result" => undefined
        }
    ]
}
```

And for a failed 3 step composite operation, where the first step succeeded and the second failed, triggering cancellation of the 3rd and rollback of the others:

```
{
    "outcome" => "failed",
    "failure-description" => "[JBAS-99999] Composite operation failed; see individual operation
results for details",
    "result" => [
        {
            "outcome" => "failed",
            "result" => {
                "name" => "Brian",
                "age" => 22
            },
            "rolled-back" => true
        },
        {
            "outcome" => "failed",
            "failure-description" => "[JBAS-12345] Some failure message",
            "rolled-back" => true
        },
        {
            "outcome" => "cancelled"
        }
    ]
}
```

Multi-Server Responses

Multi-server responses are provided by operations that target a Domain Controller or slave Host Controller and require the responder to apply the operation on multiple servers and aggregate their results (e.g. nearly all domain or host configuration updates.)

Multi-server operations are executed in several stages.

First, the operation may need to be applied against the authoritative configuration model maintained by the Domain Controller (for `domain.xml` confgurations) or a Host Controller (for a `host.xml` configuration). If there is a failure at this stage, the operation is automatically rolled back, with a response like this:

```
{
    "outcome" => "failed",
    "failure-description" => {
        "domain-failure-description" => "[JBAS-33333] Failed to apply X to the domain model"
    }
}
```

If the operation was addressed to the domain model, in the next stage the Domain Controller will ask each slave Host Controller to apply it to its local copy of the domain model. If any Host Controller fails to do so, the Domain Controller will tell all Host Controllers to revert the change, and it will revert the change locally as well. The response to the client will look like this:

```
{
    "outcome" => "failed",
    "failure-description" => {
        "host-failure-descriptions" => {
            "hostA" => "[DOM-3333] Failed to apply to the domain model",
            "hostB" => "[DOM-3333] Failed to apply to the domain model"
        }
    }
}
```

If the preceding stages succeed, the operation will be pushed to all affected servers. If the operation is successful on all servers, the response will look like this (this example operation has a void response, hence the result for each server is undefined):

```
{
    "outcome" => "success",
    "result" => undefined,
    "server-groups" => {
        "groupA" => {
            "serverA-1" => {
                "host" => "host1",
                "response" => {
                    "outcome" => "success",
                    "result" => undefined
                }
            },
            "serverA-2" => {
                "host" => "host2",
                "response" => {
                    "outcome" => "success",
                    "result" => undefined
                }
            }
        },
        "groupB" => {
            "serverB-1" => {
                "host" => "host1",
                "response" => {
                    "outcome" => "success",
                    "result" => undefined
                }
            },
            "serverB-2" => {
                "host" => "host2",
                "response" => {
                    "outcome" => "success",
                    "result" => undefined
                }
            }
        }
    }
}
```

The operation need not succeed on all servers in order to get an `"outcome" => "success"` result. All that is required is that it succeed on at least one server without the rollback policies in the rollout plan triggering a rollback on that server. An example response in such a situation would look like this:

```
{
    "outcome" => "success",
    "result" => undefined,
    "server-groups" => {
        "groupA" => {
            "serverA-1" => {
                "host" => "host1",
                "response" => {
                    "outcome" => "success",
                    "result" => undefined
                }
            },
            "serverA-2" => {
                "host" => "host2",
                "response" => {
                    "outcome" => "success",
                    "result" => undefined
                }
            }
        },
        "groupB" => {
            "serverB-1" => {
                "host" => "host1",
                "response" => {
                    "outcome" => "success",
                    "result" => undefined,
                    "rolled-back" => true
                }
            },
            "serverB-2" => {
                "host" => "host2",
                "response" => {
                    "outcome" => "success",
                    "result" => undefined,
                    "rolled-back" => true
                }
            },
            "serverB-3" => {
                "host" => "host3",
                "response" => {
                    "outcome" => "failed",
                    "failure-description" => "[DOM-4556] Something didn't work right",
                    "rolled-back" => true
                }
            }
        }
    }
}
```

Finally, if the operation fails or is rolled back on all servers, an example response would look like this:

```
{
    "outcome" => "failed",
    "server-groups" => {
        "groupA" => {
            "serverA-1" => {
                "host" => "host1",
                "response" => {
                    "outcome" => "success",
                    "result" => undefined
                }
            },
            "serverA-2" => {
                "host" => "host2",
                "response" => {
                    "outcome" => "success",
                    "result" => undefined
                }
            }
        },
        "groupB" => {
            "serverB-1" => {
                "host" => "host1",
                "response" => {
                    "outcome" => "failed",
                    "result" => undefined,
                    "rolled-back" => true
                }
            },
            "serverB-2" => {
                "host" => "host2",
                "response" => {
                    "outcome" => "failed",
                    "result" => undefined,
                    "rolled-back" => true
                }
            },
            "serverB-3" => {
                "host" => "host3",
                "response" => {
                    "outcome" => "failed",
                    "failure-description" => "[DOM-4556] Something didn't work right",
                    "rolled-back" => true
                }
            }
        }
    }
}
```

5.17.5 Description of the Management Model

A detailed description of the resources, attributes and operations that make up the management model provided by an individual WildFly instance or by any Domain Controller or slave Host Controller process can be queried using the `read-resource-description`, `read-operation-names`, `read-operation-description` and `read-child-types` operations described in the Global operations section. In this section we provide details on what's included in those descriptions.

Description of the WildFly Managed Resources

All portions of the management model exposed by WildFly are addressable via an ordered list of key/value pairs. For each addressable *Management Resource*, the following descriptive information will be available:

- `description` – String – text description of this portion of the model
- `min-occurs` – int, either 0 or 1 – Minimum number of resources of this type that must exist in a valid model. If not present, the default value is 0.
- `max-occurs` – int – Maximum number of resources of this type that may exist in a valid model. If not present, the default value depends upon the value of the final key/value pair in the address of the described resource. If this value is '*', the default value is Integer.MAX_VALUE, i.e. there is no limit. If this value is some other string, the default value is 1.
- `attributes` – Map of String (the attribute name) to complex structure – the configuration attributes available in this portion of the model. See below for the representation of each attribute.
- `operations` – Map of String (the operation name) to complex structure – the operations that can be targetted at this address. See below for the representation of each operation.
- `children` – Map of String (the type of child) to complex structure – the relationship of this portion of the model to other addressable portions of the model. See below for the representation of each child relationship.
- `head-comment-allowed` – boolean – indicates whether this portion of the model can store an XML comment that would be written in the persistent form of the model (e.g. domain.xml) before the start of the XML element that represents this portion of the model. This item is optional, and if not present defaults to true. (Note: storing XML comments in the in-memory model is not currently supported. This description key is for future use.)
- `tail-comment-allowed` – boolean – similar to `head-comment-allowed`, but indicates whether a comment just before the close of the XML element is supported. A tail comment can only be supported if the element has child elements, in which case a comment can be inserted between the final child element and the element's closing tag. This item is optional, and if not present defaults to true. (Note: storing XML comments in the in-memory model is not currently supported. This description key is for future use.)

For example:

```
{
     "description => "A manageable resource",
     "tail-comment-allowed" => false,
     "attributes" => {
          "foo" => {
               .... details of attribute foo
          }
      },
     "operations" => {
          "start" => {
               .... details of the start operation
          }
      },
     "children" => {
          "bar" => {
               .... details of the relationship with children of type "bar"
          }
      }
}
```

Description of an Attribute

An attribute is a portion of the management model that is not directly addressable. Instead, it is conceptually a property of an addressable management resource. For each attribute in the model, the following descriptive information will be available:

- `description` – String – text description of the attribute
- `type` – `org.jboss.dmr.ModelType` – the type of the attribute value. One of the enum values BIG_DECIMAL, BIG_INTEGER, BOOLEAN, BYTES, DOUBLE, INT, LIST, LONG, OBJECT, PROPERTY, STRING. Most of these are self-explanatory. An OBJECT will be represented in the detyped model as a map of string keys to values of some other legal type, conceptually similar to a `javax.management.openmbean.CompositeData`. A PROPERTY is a single key/value pair, where the key is a string, and the value is of some other legal type.
- `value-type` – ModelType or complex structure – Only present if type is LIST or OBJECT. If all elements in the LIST or all the values of the OBJECT type are of the same type, this will be one of the ModelType enums BIG_DECIMAL, BIG_INTEGER, BOOLEAN, BYTES, DOUBLE, INT, LONG, STRING. Otherwise, `value-type` will detail the structure of the attribute value, enumerating the value's fields and the type of their value. So, an attribute with a `type` of LIST and a `value-type` value of `ModelType.STRING` is analogous to a Java `List<String>`, while one with a `value-type` value of `ModelType.INT` is analogous to a Java `List<Integer>`. An attribute with a `type` of OBJECT and a `value-type` value of `ModelType.STRING` is analogous to a Java `Map<String, String>`. An attribute with a `type` of OBJECT and a `value-type` whose value is not of type `ModelType` represents a fully-defined complex object, with the object's legal fields and their values described.

- `expressions-allowed` – boolean – indicates whether the value of the attribute may be of type `ModelType.EXPRESSION`, instead of its standard type (see `type` and `value-type` above for discussion of an attribute's standard type.) A value of ModelType.EXPRESSION contains a system-property substitution expression that the server will resolve against the server-side system property map before using the value. For example, an attribute named max-threads may have an expression value of `${example.pool.max-threads:10}` instead of just 10. Default value if not present is false.
- `required` – boolean – true if the attribute must have a defined value in a representation of its portion of the model unless another attribute included in a list of `alternatives` is defined; false if it may be undefined (implying a null value) even in the absence of alternatives. If not present, true is the default.
- `nillable` – boolean – true if the attribute might not have a defined value in a representation of its portion of the model. A nillable attribute may
  be undefined either because it is not `required` or because it is required but has `alternatives` and one of the alternatives is defined.
- `storage` – String – Either "configuration" or "runtime". If "configuration", the attribute's value is stored as part of the persistent configuration (e.g. in domain.xml, host.xml or standalone.xml.) If "runtime" the attribute's value is not stored in the persistent configuration; the value only exists as long as the resource is running.
- `access-type` – String – One of "read-only", "read-write" or "metric". Whether an attribute value can be written, or can only read. A "metric" is a read-only attribute whose value is not stored in the persistent configuration, and whose value may change due to activity on the server. If an attribute is "read-write", the resource will expose an operation named "write-attribute" whose "name" parameter will accept this attribute's name and whose "value" parameter will accept a valid value for this attribute. That operation will be the standard means of updating this attribute's value.
- `restart-required` – String – One of "no-services", "all-services", "resource-services" or "jvm". Only relevant to attributes whose access-type is read-write. Indicates whether execution of a write-attribute operation whose name parameter specifies this attribute requires a restart of services (or an entire JVM) in order for the change to take effect in the runtime . See discussion of "Applying Updates to Runtime Services" below. Default value is "no-services".
- `default` – the default value for the attribute that will be used in runtime services if the attribute is not explicitly defined and no other attributes listed as `alternatives` are defined.
- `alternatives` – List of string – Indicates an exclusive relationship between attributes. If this attribute is defined, the other attributes listed in this descriptor's value should be undefined, even if their `required` descriptor says true; i.e. the presence of this attribute satisfies the requirement. Note that an attribute that is not explicitly configured but has a `default` value is still regarded as not being defined for purposes of checking whether the exclusive relationship has been violated. Default is undefined; i.e. this does not apply to most attributes.
- `requires` – List of string – Indicates that if this attribute has a value (other than undefined), the other attributes listed in this descriptor's value must also have a value, even if their required descriptor says false. This would typically be used in conjunction with alternatives. For example, attributes "a" and "b" are required, but are alternatives to each other; "c" and "d" are optional. But "b" requires "c" and "d", so if "b" is used, "c" and "d" must also be defined. Default is undefined; i.e. this does not apply to most attributes.

- `capability-reference` – string – if defined indicates that this attribute's value specifies the dynamic portion of the name of the specified capability provided by another resource. This indicates the attribute is a reference to another area of the management model. (Note that at present some attributes that reference other areas of the model may not provide this information.)
- `head-comment-allowed` – boolean – indicates whether the model can store an XML comment that would be written in the persistent form of the model (e.g. domain.xml) before the start of the XML element that represents this attribute. This item is optional, and if not present defaults to false. (This is a different default from what is used for an entire management resource, since model attributes often map to XML attributes, which don't allow comments.) (Note: storing XML comments in the in-memory model is not currently supported. This description key is for future use.)
- `tail-comment-allowed` – boolean – similar to head-comment-allowed, but indicates whether a comment just before the close of the XML element is supported. A tail comment can only be supported if the element has child elements, in which case a comment can be inserted between the final child element and the element's closing tag. This item is optional, and if not present defaults to false. (This is a different default from what is used for an entire management resource, since model attributes often map to XML attributes, which don't allow comments.) (Note: storing XML comments in the in-memory model is not currently supported. This description key is for future use.)
- arbitrary key/value pairs that further describe the attribute value, e.g. "max" => 2. See "Arbitrary Descriptors" below.

Some examples:

```
"foo" => {
     "description" => "The foo",
     "type" => INT,
     "max" => 2
}
```

```
"bar" => {
     "description" => "The bar",
     "type" => OBJECT,
     "value-type" => {
          "size" => INT,
          "color" => STRING
     }
}
```

Description of an Operation

A management resource may have operations associated with it. The description of an operation will include the following information:

- `operation-name` – String – the name of the operation
- `description` – String – text description of the operation
- `request-properties` – Map of String to complex structure – description of the parameters of the operation. Keys are the names of the parameters, values are descriptions of the parameter value types. See below for details on the description of parameter value types.
- `reply-properties` – complex structure, or empty – description of the return value of the operation, with an empty node meaning void. See below for details on the description of operation return value types.
- `restart-required` – String – One of "no-services", "all-services", "resource-services" or "jvm". Indicates whether the operation makes a configuration change that requires a restart of services (or an entire JVM) in order for the change to take effect in the runtime. See discussion of "Applying Updates to Runtime Services" below. Default value is "no-services".

Description of an Operation Parameter or Return Value

- `description` – String – text description of the parameter or return value
- `type` – `org.jboss.dmr.ModelType` – the type of the parameter or return value. One of the enum values BIG_DECIMAL, BIG_INTEGER, BOOLEAN, BYTES, DOUBLE, INT, LIST, LONG, OBJECT, PROPERTY, STRING.
- `value-type` – `ModelType` or complex structure – Only present if type is LIST or OBJECT. If all elements in the LIST or all the values of the OBJECT type are of the same type, this will be one of the ModelType enums BIG_DECIMAL, BIG_INTEGER, BOOLEAN, BYTES, DOUBLE, INT, LIST, LONG, PROPERTY, STRING. Otherwise, value-type will detail the structure of the attribute value, enumerating the value's fields and the type of their value.So, a parameter with a `type` of LIST and a `value-type` value of `ModelType.STRING` is analogous to a Java `List<String>`, while one with a `value-type` value of `ModelType.INT` is analogous to a Java `List<Integer>`. A parameter with a `type` of OBJECT and a `value-type` value of `ModelType.STRING` is analogous to a Java `Map<String, String>`. A parameter with a `type` of OBJECT and a `value-type` whose value is not of type `ModelType` represents a fully-defined complex object, with the object's legal fields and their values described.
- `expressions-allowed` – boolean – indicates whether the value of the the parameter or return value may be of type `ModelType.EXPRESSION`, instead its standard type (see type and value-type above for discussion of the standard type.) A value of ModelType.EXPRESSION contains a system-property substitution expression that the server will resolve against the server-side system property map before using the value. For example, a parameter named max-threads may have an expression value of `${example.pool.max-threads:10}` instead of just 10. Default value if not present is false.
- `required` – boolean – true if the parameter or return value must have a defined value in the operation or response unless another item included in a list of `alternatives` is defined; false if it may be undefined (implying a null value) even in the absence of alternatives. If not present, true is the default.

- `nillable` – boolean – true if the parameter or return value might not have a defined value in a representation of its portion of the model. A nillable parameter or return value may be undefined either because it is not `required` or because it is required but has `alternatives` and one of the alternatives is defined.
- `default` – the default value for the parameter that will be used in runtime services if the parameter is not explicitly defined and no other parameters listed as `alternatives` are defined.
- `restart-required` – String – One of "no-services", "all-services", "resource-services" or "jvm". Only relevant to attributes whose access-type is read-write. Indicates whether execution of a write-attribute operation whose name parameter specifies this attribute requires a restart of services (or an entire JVM) in order for the change to take effect in the runtime . See discussion of "Applying Updates to Runtime Services" below. Default value is "no-services".
- `alternatives` – List of string – Indicates an exclusive relationship between parameters. If this attribute is defined, the other parameters listed in this descriptor's value should be undefined, even if their required descriptor says true; i.e. the presence of this parameter satisfies the requirement. Note that an parameer that is not explicitly configured but has a `default` value is still regarded as not being defined for purposes of checking whether the exclusive relationship has been violated. Default is undefined; i.e. this does not apply to most parameters.
- `requires` – List of string – Indicates that if this parameter has a value (other than undefined), the other parameters listed in this descriptor's value must also have a value, even if their required descriptor says false. This would typically be used in conjunction with alternatives. For example, parameters "a" and "b" are required, but are alternatives to each other; "c" and "d" are optional. But "b" requires "c" and "d", so if "b" is used, "c" and "d" must also be defined. Default is undefined; i.e. this does not apply to most parameters.
- arbitrary key/value pairs that further describe the attribute value, e.g. "max" =>2. See "Arbitrary Descriptors" below.

Arbitrary Descriptors

The description of an attribute, operation parameter or operation return value type can include arbitrary key/value pairs that provide extra information. Whether a particular key/value pair is present depends on the context, e.g. a pair with key "max" would probably only occur as part of the description of some numeric type.

Following are standard keys and their expected value type. If descriptor authors want to add an arbitrary key/value pair to some descriptor and the semantic matches the meaning of one of the following items, the standard key/value type must be used.

- `min` – int – the minimum value of some numeric type. The absence of this item implies there is no minimum value.
- `max` – int – the maximum value of some numeric type. The absence of this item implies there is no maximum value.
- `min-length` – int – the minimum length of some string, list or byte[] type. The absence of this item implies a minimum length of zero.
- `max-length` – int – the maximum length of some string, list or byte[]. The absence of this item implies there is no maximum value.
- `allowed` – List – a list of legal values. The type of the elements in the list should match the type of the attribute.
- `unit` - The unit of the value, if one is applicable - e.g. ns, ms, s, m, h, KB, MB, TB. See the `org.jboss.as.controller.client.helpers.MeasurementUnit` in the org.jboss.as:jboss-as-controller-client artifact for a listing of legal measurement units..

Some examples:

```
{
     "operation-name" => "incrementFoo",
     "description" => "Increase the value of the 'foo' attribute by the given amount",
     "request-properties" => {
          "increment" => {
               "type" => INT,
               "description" => "The amount to increment",
               "required" => true
     }},
     "reply-properties" => {
               "type" => INT,
               "description" => "The new value",
     }
}
```

```
{
     "operation-name" => "start",
     "description" => "Starts the thing",
     "request-properties" => {},
     "reply-properties" => {}
}
```

Description of Parent/Child Relationships

The address used to target an addressable portion of the model must be an ordered list of key value pairs. The effect of this requirement is the addressable portions of the model naturally form a tree structure, with parent nodes in the tree defining what the valid keys are and the children defining what the valid values are. The parent node also defines the cardinality of the relationship. The description of the parent node includes a children element that describes these relationships:

```
{
     ....
     "children" => {
          "connector" => {
               .... description of the relationship with children of type "connector"
          },
          "virtual-host" => {
               .... description of the relationship with children of type "virtual-host"
          }
}
```

The description of each relationship will include the following elements:

- `description` – String – text description of the relationship
- `model-description` – either "undefined" or a complex structure – This is a node of ModelType.OBJECT, the keys of which are legal values for the value portion of the address of a resource of this type, with the special character '*' indicating the value portion can have an arbitrary value. The values in the node are the full description of the particular child resource (its text description, attributes, operations, children) as detailed above. This `model-description` may also be "undefined", i.e. a null value, if the query that asked for the parent node's description did not include the "recursive" param set to true.

Example with if the recursive flag was set to true:

```
{
     "description" => "The connectors used to handle client connections",
     "model-description" => {
          "*" => {
              "description" => "Handles client connections",
              "min-occurs" => 1,
              "attributes => {
                   ... details of children as documented above
              },
              "operations" => {
                   .... details of operations as documented above
              },
              "children" => {
                   .... details of the children's children
              }
          }
     }
}
```

If the recursive flag was false:

```
{
     "description" => "The connectors used to handle client connections",
     "model-description" => undefined
}
```

Applying Updates to Runtime Services

An attribute or operation description may include a "`restart-required`" descriptor; this section is an explanation of the meaning of that descriptor.

An operation that changes a management resource's persistent configuration usually can also also affect a runtime service associated with the resource. For example, there is a runtime service associated with any host.xml or standalone.xml `<interface>` element; other services in the runtime depend on that service to provide the `InetAddress` associated with the interface. In many cases, an update to a resource's persistent configuration can be immediately applied to the associated runtime service. The runtime service's state is updated to reflect the new value(s).

However, in many cases the runtime service's state cannot be updated without restarting the service. Restarting a service can have broad effects. A restart of a service A will trigger a restart of other services B, C and D that depend A, triggering a restart of services that depend on B, C and D, etc. Those service restarts may very well disrupt handling of end-user requests.

Because restarting a service can be disruptive to end-user request handling, the handlers for management operations will not restart any service without some form of explicit instruction from the end user indicating a service restart is desired. In a few cases, simply executing the operation is an indication the user wants services to restart (e.g. a `/host=master/server-config=server-one:restart` operation in a managed domain, or a `/:reload` operation on a standalone server.) For all other cases, if an operation (or attribute write) cannot be performed without restarting a service, the metadata describing the operation or attribute will include a "`restart-required`" descriptor whose value indicates what is necessary for the operation to affect the runtime:

- `no-services` – Applying the operation to the runtime does not require the restart of any services. This value is the default if the restart-required descriptor is not present.
- `all-services` – The operation can only immediately update the persistent configuration; applying the operation to the runtime will require a subsequent restart of all services in the affected VM. Executing the operation will put the server into a "`reload-required`" state. Until a restart of all services is performed the response to this operation and to any subsequent operation will include a response header "`process-state" => "reload-required`". For a standalone server, a restart of all services can be accomplished by executing the `/:reload` CLI command. For a server in a managed domain, restarting all services currently requires a full restart of the affected server VM (e.g. `/host=master/server-config=server-one:restart`).
- `jvm` --The operation can only immediately update the persistent configuration; applying the operation to the runtime will require a full process restart (i.e. stop the JVM and launch a new JVM). Executing the operation will put the server into a "`restart-required`" state. Until a restart is performed the response to this operation and to any subsequent operation will include a response header "`process-state" => "restart-required`". For a standalone server, a full process restart requires first stopping the server via OS-level operations (Ctrl-C, kill) or via the `/:shutdown` CLI command, and then starting the server again from the command line. For a server in a managed domain, restarting a server requires executing the `/host=<host>/server-config=<server>:restart` operation.
- `resource-services` – The operation can only immediately update the persistent configuration; applying the operation to the runtime will require a subsequent restart of some services associated with the resource. If the operation includes the request header `"allow-resource-service-restart" => true`, the handler for the operation will go ahead and restart the runtime service. Otherwise executing the operation will put the server into a "`reload-required`" state. (See the discussion of "`all-services`" above for more on the "`reload-required`" state.)

5.17.6 Detyped management and the jboss-dmr library

The management model exposed by WildFly is very large and complex. There are dozens, probably hundreds of logical concepts involved – hosts, server groups, servers, subsystems, datasources, web connectors, and on and on – each of which in a classic objected oriented API design could be represented by a Java *type* (i.e. a Java class or interface.) However, a primary goal in the development of WildFly's native management API was to ensure that clients built to use the API had as few compile-time and run-time dependencies on JBoss-provided classes as possible, and that the API exposed by those libraries be powerful but also simple and stable. A management client running with the management libraries created for an earlier version of WildFly should still work if used to manage a later version domain. The management client libraries needed to be *forward compatible*.

It is highly unlikely that an API that consists of hundreds of Java types could be kept forward compatible. Instead, the WildFly management API is a *detyped* API. A detyped API is like decaffeinated coffee – it still has a little bit of caffeine, but not enough to keep you awake at night. WildFly's management API still has a few Java types in it (it's impossible for a Java library to have no types!) but not enough to keep you (or us) up at night worrying that your management clients won't be forward compatible.

A detyped API works by making it possible to build up arbitrarily complex data structures using a small number of Java types. All of the parameter values and return values in the API are expressed using those few types. Ideally, most of the types are basic JDK types, like `java.lang.String`, `java.lang.Integer`, etc. In addition to the basic JDK types, WildFly's detyped management API uses a small library called **jboss-dmr**. The purpose of this section is to provide a basic overview of the jboss-dmr library.

Even if you don't use jboss-dmr directly (probably the case for all but a few users), some of the information in this section may be useful. When you invoke operations using the application server's Command Line Interface, the return values are just the text representation of of a jboss-dmr `ModelNode`. If your CLI commands require complex parameter values, you may yourself end up writing the text representation of a `ModelNode`. And if you use the HTTP management API, all response bodies as well as the request body for any POST will be a JSON representation of a `ModelNode`.

The source code for jboss-dmr is available on Github. The maven coordinates for a jboss-dmr release are `org.jboss.jboss-dmr:jboss-dmr`.

ModelNode and ModelType

The public API exposed by jboss-dmr is very simple: just three classes, one of which is an enum!

The primary class is `org.jboss.dmr.ModelNode`. A `ModelNode` is essentially just a wrapper around some *value*; the value is typically some basic JDK type. A `ModelNode` exposes a `getType()` method. This method returns a value of type `org.jboss.dmr.ModelType`, which is an enum of all the valid types of values. And that's 95% of the public API; a class and an enum. (We'll get to the third class, `Property`, below.)

Basic ModelNode manipulation

To illustrate how to work with `ModelNode` s, we'll use the Beanshell scripting library. We won't get into many details of beanshell here; it's a simple and intuitive tool and hopefully the following examples are as well.

We'll start by launching a beanshell interpreter, with the jboss-dmr library available on the classpath. Then we'll tell beanshell to import all the jboss-dmr classes so they are available for use:

```
$ java -cp bsh-2.0b4.jar:jboss-dmr-1.0.0.Final.jar bsh.Interpreter
BeanShell 2.0b4 - by Pat Niemeyer (pat@pat.net)
bsh % import org.jboss.dmr.*;
bsh %
```

Next, create a `ModelNode` and use the beanshell `print` function to output what type it is:

```
bsh % ModelNode node = new ModelNode();
bsh % print(node.getType());
UNDEFINED
```

A new `ModelNode` has no value stored, so its type is `ModelType.UNDEFINED`.

Use one of the overloaded `set` method variants to assign a node's value:

```
bsh % node.set(1);
bsh % print(node.getType());
INT
bsh % node.set(true);
bsh % print(node.getType());
BOOLEAN
bsh % node.set("Hello, world");
bsh % print(node.getType());
STRING
```

Use one of the `asXXX()` methods to retrieve the value:

```
bsh % node.set(2);
bsh % print(node.asInt());
2
bsh % node.set("A string");
bsh % print(node.asString());
A string
```

`ModelNode` will attempt to perform type conversions when you invoke the `asXXX` methods:

```
bsh % node.set(1);
bsh % print(node.asString());
1
bsh % print(node.asBoolean());
true
bsh % node.set(0);
bsh % print(node.asBoolean());
false
bsh % node.set("true");
bsh % print(node.asBoolean());
true
```

Not all type conversions are possible:

```
bsh % node.set("A string");
bsh % print(node.asInt());
// Error: // Uncaught Exception: Method Invocation node.asInt : at Line: 20 : in file: <unknown
file> : node .asInt ( )

Target exception: java.lang.NumberFormatException: For input string: "A string"

java.lang.NumberFormatException: For input string: "A string"
 at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
 at java.lang.Integer.parseInt(Integer.java:449)
 at java.lang.Integer.parseInt(Integer.java:499)
 at org.jboss.dmr.StringModelValue.asInt(StringModelValue.java:61)
 at org.jboss.dmr.ModelNode.asInt(ModelNode.java:117)
        ....
```

The `ModelNode.getType()` method can be used to ensure a node has an expected value type before attempting a type conversion.

One `set` variant takes another `ModelNode` as its argument. The value of the passed in node is copied, so there is no shared state between the two model nodes:

```
bsh % node.set("A string");
bsh % ModelNode another = new ModelNode();
bsh % another.set(node);
bsh % print(another.asString());
A string
bsh % node.set("changed");
bsh % print(node.asString());
changed
bsh % print(another.asString());
A string
```

A `ModelNode` can be cloned. Again, there is no shared state between the original node and its clone:

```
bsh % ModelNode clone = another.clone();
bsh % print(clone.asString());
A string
bsh % another.set(42);
bsh % print(another.asString());
42
bsh % print(clone.asString());
A string
```

Use the `protect()` method to make a `ModelNode` immutable:

```
bsh % clone.protect();
bsh % clone.set("A different string");
// Error: // Uncaught Exception: Method Invocation clone.set : at Line: 15 : in file: <unknown
file> : clone .set ( "A different string" )

Target exception: java.lang.UnsupportedOperationException

java.lang.UnsupportedOperationException
 at org.jboss.dmr.ModelNode.checkProtect(ModelNode.java:1441)
 at org.jboss.dmr.ModelNode.set(ModelNode.java:351)
        ....
```

Lists

The above examples aren't particularly interesting; if all we can do with a `ModelNode` is wrap a simple Java primitive, what use is that? However, a `ModelNode`'s value can be more complex than a simple primitive, and using these more complex types we can build complex data structures. The first more complex type is `ModelType.LIST`.

Use the `add` methods to initialize a node's value as a list and add to the list:

```
bsh % ModelNode list = new ModelNode();
bsh % list.add(5);
bsh % list.add(10);
bsh % print(list.getType());
LIST
```

Use `asInt()` to find the size of the list:

```
bsh % print(list.asInt());
2
```

Use the overloaded `get` method variant that takes an int param to retrieve an item. The item is returned as a `ModelNode`:

```
bsh % ModelNode child = list.get(1);
bsh % print(child.asInt());
10
```

Elements in a list need not all be of the same type:

```
bsh % list.add("A string");
bsh % print(list.get(1).getType());
INT
bsh % print(list.get(2).getType());
STRING
```

Here's one of the trickiest things about jboss-dmr: *The* `get` *methods actually mutate state; they are not "read-only"*. For example, calling `get` with an index that does not exist yet in the list will actually create a child of type `ModelType.UNDEFINED` at that index (and will create UNDEFINED children for any intervening indices.)

```
bsh % ModelNode four = list.get(4);
bsh % print(four.getType());
UNDEFINED
bsh % print(list.asInt());
6
```

Since the `get` call always returns a `ModelNode` and never `null` it is safe to manipulate the return value:

```
bsh % list.get(5).set(30);
bsh % print(list.get(5).asInt());
30
```

That's not so interesting in the above example, but later on with node of type `ModelType.OBJECT` we'll see how that kind of method chaining can let you build up fairly complex data structures with a minimum of code.

Use the `asList()` method to get a `List<ModelNode>` of the children:

```
bsh % for (ModelNode element : list.asList()) {
print(element.getType());
}
INT
INT
STRING
UNDEFINED
UNDEFINED
INT
```

The `asString()` and `toString()` methods provide slightly differently formatted text representations of a `ModelType.LIST` node:

```
bsh % print(list.asString());
[5,10,"A string",undefined,undefined,30]
bsh % print(list.toString());
[
    5,
    10,
    "A string",
    undefined,
    undefined,
    30
]
```

Finally, if you've previously used `set` to assign a node's value to some non-list type, you cannot use the `add` method:

```
bsh % node.add(5);
// Error: // Uncaught Exception: Method Invocation node.add : at Line: 18 : in file: <unknown
file> : node .add ( 5 )

Target exception: java.lang.IllegalArgumentException

java.lang.IllegalArgumentException
 at org.jboss.dmr.ModelValue.addChild(ModelValue.java:120)
 at org.jboss.dmr.ModelNode.add(ModelNode.java:1007)
 at org.jboss.dmr.ModelNode.add(ModelNode.java:761)
        ...
```

You can, however, use the `setEmptyList()` method to change the node's type, and then use `add`:

```
bsh % node.setEmptyList();
bsh % node.add(5);
bsh % print(node.toString());
[5]
```

Properties

The third public class in the jboss-dmr library is `org.jboss.dmr.Property`. A `Property` is a `String => ModelNode` tuple.

```
bsh % Property prop = new Property("stuff", list);
bsh % print(prop.toString());
org.jboss.dmr.Property@79a5f739
bsh % print(prop.getName());
stuff
bsh % print(prop.getValue());
[
    5,
    10,
    "A string",
    undefined,
    undefined,
    30
]
```

The property can be passed to `ModelNode.set`:

```
bsh % node.set(prop);
bsh % print(node.getType());
PROPERTY
```

The text format for a node of `ModelType.PROPERTY` is:

```
bsh % print(node.toString());
("stuff" => [
    5,
    10,
    "A string",
    undefined,
    undefined,
    30
])
```

Directly instantiating a `Property` via its constructor is not common. More typically one of the two argument `ModelNode.add` or `ModelNode.set` variants is used. The first argument is the property name:

```
bsh % ModelNode simpleProp = new ModelNode();
bsh % simpleProp.set("enabled", true);
bsh % print(simpleProp.toString());
("enabled" => true)
bsh % print(simpleProp.getType());
PROPERTY
bsh % ModelNode propList = new ModelNode();
bsh % propList.add("min", 1);
bsh % propList.add("max", 10);
bsh % print(propList.toString());
[
    ("min" => 1),
    ("max" => 10)
]
bsh % print(propList.getType());
LIST
bsh % print(propList.get(0).getType());
PROPERTY
```

The `asPropertyList()` method provides easy access to a `List<Property>`:

```
bsh % for (Property prop : propList.asPropertyList()) {
print(prop.getName() + " = " + prop.getValue());
}
min = 1
max = 10
```

ModelType.OBJECT

The most powerful and most commonly used complex value type in jboss-dmr is `ModelType.OBJECT`. A `ModelNode` whose value is `ModelType.OBJECT` internally maintains a `Map<String, ModelNode`.

Use the `get` method variant that takes a string argument to add an entry to the map. If no entry exists under the given name, a new entry is added with a the value being a `ModelType.UNDEFINED` node. The node is returned:

```
bsh % ModelNode range = new ModelNode();
bsh % ModelNode min = range.get("min");
bsh % print(range.toString());
{"min" => undefined}
bsh % min.set(2);
bsh % print(range.toString());
{"min" => 2}
```

Again **it is important to remember that the `get` operation may mutate the state of a model node by adding a new entry.** *It is not a read-only operation.*

Since `get` will never return `null`, a common pattern is to use method chaining to create the key/value pair:

```
bsh % range.get("max").set(10);
bsh % print(range.toString());
{
    "min" => 2,
    "max" => 10
}
```

A call to `get` passing an already existing key will of course return the same model node as was returned the first time `get` was called with that key:

```
bsh % print(min == range.get("min"));
true
```

Multiple parameters can be passed to `get`. This is a simple way to traverse a tree made up of `ModelType.OBJECT` nodes. Again, `get` may mutate the node on which it is invoked; e.g. it will actually create the tree if nodes do not exist. This next example uses a workaround to get beanshell to handle the overloaded `get` method that takes a variable number of arguments:

```
bsh % String[] varargs = { "US", "Missouri", "St. Louis" };
bsh % salesTerritories.get(varargs).set("Brian");
bsh % print(salesTerritories.toString());
{"US" => {"Missouri" => {"St. Louis" => "Brian"}}}
```

The normal syntax would be:

```
salesTerritories.get("US", "Missouri", "St. Louis").set("Brian");
```

The key/value pairs in the map can be accessed as a `List<Property>`:

```
bsh % for (Property prop : range.asPropertyList()) {
print(prop.getName() + " = " + prop.getValue());
}
min = 2
```

The semantics of the backing map in a node of `ModelType.OBJECT` are those of a `LinkedHashMap`. The map remembers the order in which key/value pairs are added. This is relevant when iterating over the pairs after calling `asPropertyList()` and for controlling the order in which key/value pairs appear in the output from `toString()`.

Since the `get` method will actually mutate the state of a node if the given key does not exist, `ModelNode` provides a couple methods to let you check whether the entry is there. The `has` method simply does that:

```
bsh % print(range.has("unit"));
false
bsh % print(range.has("min"));
true
```

Very often, the need is to not only know whether the key/value pair exists, but whether the value is defined (i.e. not `ModelType.UNDEFINED`. This kind of check is analogous to checking whether a field in a Java class has a null value. The `hasDefined` lets you do this:

```
bsh % print(range.hasDefined("unit"));
false
bsh % // Establish an undefined child 'unit';
bsh % range.get("unit");
bsh % print(range.toString());
{
    "min" => 2,
    "max" => 10,
    "unit" => undefined
}
bsh % print(range.hasDefined("unit"));
false
bsh % range.get("unit").set("meters");
bsh % print(range.hasDefined("unit"));
true
```

ModelType.EXPRESSION

A value of type `ModelType.EXPRESSION` is stored as a string, but can later be *resolved* to different value. The string has a special syntax that should be familiar to those who have used the system property substitution feature in previous JBoss AS releases.

```
[<prefix>][${<system-property-name>[:<default-value>]}][<suffix>]*
```

For example:

```
${queue.length}
http://${host}
http://${host:localhost}:${port:8080}/index.html
```

Use the `setExpression` method to set a node's value to type expression:

```
bsh % ModelNode expression = new ModelNode();
bsh % expression.setExpression("${queue.length}");
bsh % print(expression.getType());
EXPRESSION
```

Calling `asString()` returns the same string that was input:

```
bsh % print(expression.asString());
${queue.length}
```

However, calling `toString()` tells you that this node's value is not of `ModelType.STRING`:

```
bsh % print(expression.toString());
expression "${queue.length}"
```

When the `resolve` operation is called, the string is parsed and any embedded system properties are resolved against the JVM's current system property values. A new `ModelNode` is returned whose value is the resolved string:

```
bsh % System.setProperty("queue.length", "10");
bsh % ModelNode resolved = expression.resolve();
bsh % print(resolved.asInt());
10
```

Note that the type of the `ModelNode` returned by `resolve()` is `ModelType.STRING`:

```
bsh % print(resolved.getType());
STRING
```

The `resolved.asInt()` call in the previous example only worked because the string "10" happens to be convertible into the int 10.

Calling `resolve()` has no effect on the value of the node on which the method is invoked:

```
bsh % resolved = expression.resolve();
bsh % print(resolved.toString());
"10"
bsh % print(expression.toString());
expression "${queue.length}"
```

If an expression cannot be resolved, `resolve` just uses the original string. The string can include more than one system property substitution:

```
bsh % expression.setExpression("http://${host}:${port}/index.html");
bsh % resolved = expression.resolve();
bsh % print(resolved.asString());
http://${host}:${port}/index.html
```

The expression can optionally include a default value, separated from the name of the system property by a colon:

```
bsh % expression.setExpression("http://${host:localhost}:${port:8080}/index.html");
bsh % resolved = expression.resolve();
bsh % print(resolved.asString());
http://localhost:8080/index.html
```

Actually including a system property substitution in the expression is not required:

```
bsh % expression.setExpression("no system property");
bsh % resolved = expression.resolve();
bsh % print(resolved.asString());
no system property
bsh % print(expression.toString());
expression "no system property"
```

The `resolve` method works on nodes of other types as well; it returns a copy without attempting any real resolution:

```
bsh % ModelNode basic = new ModelNode();
bsh % basic.set(10);
bsh % resolved = basic.resolve();
bsh % print(resolved.getType());
INT
bsh % resolved.set(5);
bsh % print(resolved.asInt());
5
bsh % print(basic.asInt());
10
```

ModelType.TYPE

You can also pass one of the values of the `ModelType` enum to `set`:

```
bsh % ModelNode type = new ModelNode();
bsh % type.set(ModelType.LIST);
bsh % print(type.getType());
TYPE
bsh % print(type.toString());
LIST
```

This is useful when using a `ModelNode` data structure to describe another `ModelNode` data structure.

Full list of ModelNode types

BIG_DECIMAL
BIG_INTEGER
BOOLEAN
BYTES
DOUBLE
EXPRESSION
INT
LIST
LONG
OBJECT
PROPERTY
STRING
TYPE
UNDEFINED

Text representation of a ModelNode

TODO – document the grammar

JSON representation of a ModelNode

TODO – document the grammar

5.17.7 Global operations

The WildFly management API includes a number of operations that apply to every resource.

The read-resource operation

Reads a management resource's attribute values along with either basic or complete information about any child resources. Supports the
following parameters, none of which are required:

- `recursive` – (boolean, default is `false`) – whether to include complete information about child resources, recursively.
- `recursive-depth` – (int) – The depth to which information about child resources should be included if `recursive` is `true`. If not set, the depth will be unlimited; i.e. all descendant resources will be included.
- `proxies` – (boolean, default is `false`) – whether to include remote resources in a recursive query (i.e. host level resources from slave Host Controllers in a query of the Domain Controller; running server resources in a query of a host).
- `include-runtime` – (boolean, default is `false`) – whether to include runtime attributes (i.e. those whose value does not come from the persistent configuration) in the response.
- `include-defaults` – (boolean, default is `true`) – whether to include in the result default values not set by users. Many attributes have a default value that will be used in the runtime if the users have not provided an explicit value. If this parameter is `false` the value for such attributes in the result will be `undefined`. If `true` the result will include the default value for such parameters.

The read-attribute operation

Reads the value of an individual attribute. Takes a single, required, parameter:

- `name` – (string) – the name of the attribute to read.
- `include-defaults` – (boolean, default is `true`) – whether to include in the result default values not set by users. Many attributes have a default value that will be used in the runtime if the users have not provided an explicit value. If this parameter is `false` the value for such attributes in the result will be `undefined`. If `true` the result will include the default value for such parameters.

The write-attribute operation

Writes the value of an individual attribute. Takes two required parameters:

- `name` – (string) – the name of the attribute to write.
- `value` – (type depends on the attribute being written) – the new value.

The undefine-attribute operation

Sets the value of an individual attribute to the `undefined` value, if such a value is allowed for the attribute. The operation will fail if the `undefined` value is not allowed. Takes a single required parameter:

- `name` – (string) – the name of the attribute to write.

The list-add operation

Adds an element to the value of a list attribute, adding the element to the end of the list unless the optional attribute `index` is passed:

- `name` – (string) – the name of the list attribute to add new value to.
- `value` – (type depends on the element being written) – the new element to be added to the attribute value.
- `index` – (int, optional) – index where in the list to add the new element. By default it is `undefined` meaning add at the end. Index is zero based.

This operation will fail if the specified attribute is not a list.

The list-remove operation

Removes an element from the value of a list attribute, either the element at a specified `index`, or the first element whose value matches a specified `value`:

- `name` – (string) – the name of the list attribute to add new value to.
- `value` – (type depends on the element being written, optional) – the element to be removed. Optional and ignored if `index` is specified.
- `index` – (int, optional) – index in the list whose element should be removed. By default it is `undefined`, meaning `value` should be specified.

This operation will fail if the specified attribute is not a list.

The list-get operation

Gets one element from a list attribute by its index

- `name` – (string) – the name of the list attribute
- `index` – (int, required) – index of element to get from list

This operation will fail if the specified attribute is not a list.

The list-clear operation

Empties the list attribute. It is different from `:undefine-attribute` as it results in attribute of type list with 0 elements, whereas `:undefine-attribute` results in an `undefined` value for the attribute

- `name` – (string) – the name of the list attribute

This operation will fail if the specified attribute is not a list.

The map-put operation

Adds an key/value pair entry to the value of a map attribute:

- `name` – (string) – the name of the map attribute to add the new entry to.
- `key` – (string) – the key of the new entry to be added.
- `value` – (type depends on the entry being written) – the value of the new entry to be added to the attribute value.

This operation will fail if the specified attribute is not a map.

The map-remove operation

Removes an entry from the value of a map attribute:

- `name` – (string) – the name of the map attribute to remove the new entry from.
- `key` – (string) – the key of the entry to be removed.

This operation will fail if the specified attribute is not a map.

The map-get operation

Gets the value of one entry from a map attribute

- `name` – (string) – the name of the map attribute
- `key` – (string) – the key of the entry.

This operation will fail if the specified attribute is not a map.

The map-clear operation

Empties the map attribute. It is different from `:undefine-attribute` as it results in attribute of type map with 0 entries, whereas `:undefine-attribute` results in an `undefined` value for the attribute

- `name` – (string) – the name of the map attribute

This operation will fail if the specified attribute is not a map.

The read-resource-description operation

Returns the description of a resource's attributes, types of children and, optionally, operations. Supports the following parameters, none of which are required:

- `recursive` – (boolean, default is `false`) – whether to include information about child resources, recursively.
- `proxies` – (boolean, default is `false`) – whether to include remote resources in a recursive query (i.e. host level resources from slave Host Controllers in a query of the Domain Controller; running server resources in a query of a host)
- `operations` – (boolean, default is `false`) – whether to include descriptions of the resource's operations
- `inherited` – (boolean, default is `true`) – if `operations` is `true`, whether to include descriptions of operations inherited from higher level resources. The global operations described in this section are themselves inherited from the root resource, so the primary effect of setting `inherited` to `false` is to exclude the descriptions of the global operations from the output.

See Description of the Management Model for details on the result of this operation.

The read-operation-names operation

Returns a list of the names of all the operations the resource supports. Takes no parameters.

The read-operation-description operation

Returns the description of an operation, along with details of its parameter types and its return value. Takes a single, required, parameter:

- `name` – (string) – the name of the operation

See Description of the Management Model for details on the result of this operation.

The read-children-types operation

Returns a list of the types of child resources the resource supports. Takes two optional parameters:

- `include-aliases` – (boolean, default is `false`) – whether to include alias children (i.e. those which are aliases of other sub-resources) in the response.
- `include-singletons` – (boolean, default is `false`) – whether to include singleton children (i.e. those are children that acts as resource aggregate and are registered with a wildcard name) in the response wildfly-dev discussion around this topic.

The read-children-names operation

Returns a list of the names of all child resources of a given type. Takes a single, required, parameter:

- `child-type` – (string) – the name of the type

The read-children-resources operation

Returns information about all of a resource's children that are of a given type. For each child resource, the returned information is equivalent to executing the `read-resource` operation on that resource. Takes the following parameters, of which only {{child-type} is required:

- `child-type` – (string) – the name of the type of child resource
- `recursive` – (boolean, default is `false`) – whether to include complete information about child resources, recursively.
- `recursive-depth` – (int) – The depth to which information about child resources should be included if `recursive` is {{true}. If not set, the depth will be unlimited; i.e. all descendant resources will be included.
- `proxies` – (boolean, default is `false`) – whether to include remote resources in a recursive query (i.e. host level resources from slave Host Controllers in a query of the Domain Controller; running server resources in a query of a host)
- `include-runtime` – (boolean, default is `false`) – whether to include runtime attributes (i.e. those whose value does not come from the persistent configuration) in the response.
- `include-defaults` – (boolean, default is `true`) – whether to include in the result default values not set by users. Many attributes have a default value that will be used in the runtime if the users have not provided an explicit value. If this parameter is `false` the value for such attributes in the result will be `undefined`. If `true` the result will include the default value for such parameters.

The read-attribute-group operation

Returns a list of attributes of a type for a given attribute group name. For each attribute the returned information is equivalent to executing the `read-attribute` operation of that resource. Takes the following parameters, of which only {{name} is required:

- `name` – (string) – the name of the attribute group to read.
- `include-defaults` – (boolean, default is `true`) – whether to include in the result default values not set by users. Many attributes have a default value that will be used in the runtime if the users have not provided an explicit value. If this parameter is `false` the value for such attributes in the result will be `undefined`. If `true` the result will include the default value for such parameters.
- `include-runtime` – (boolean, default is `false`) – whether to include runtime attributes (i.e. those whose value does not come from the persistent configuration) in the response.
- `include-aliases` – (boolean, default is `false`) – whether to include alias attributes (i.e. those which are alias of other attributes) in the response.

The read-attribute-group-names operation

Returns a list of attribute groups names for a given type. Takes no parameters.

Standard Operations

Besides the global operations described above, by convention nearly every resource should expose an `add` operation and a `remove` operation. Exceptions to this convention are the root resource, and resources that do not store persistent configuration and are created dynamically at runtime (e.g. resources representing the JVM's platform mbeans or resources representing aspects of the running state of a deployment.)

The add operation

The operation that creates a new resource must be named `add`. The operation may take zero or more parameters; what those parameters are depends on the resource being created.

The remove operation

The operation that removes an existing resource must be named `remove`. The operation should take no parameters.

5.17.8 The HTTP management API

Introduction

The Management API in WildFly is accessible through multiple channels, one of them being HTTP and JSON.

Even if you haven't used a curl command line you might already have used this channel since it is how the web console interact with the Management API.

WildFly 9 is distributed secured by default, the default security mechanism is *username / password* based making use of **HTTP Digest** for the authentication process.

Thus **you need to create a user** with the add-user.sh script.

Interacting with the model

Since we must be authenticated , the client will have to support HTTP Digest authentication.

For example this can be activated in *curl* using the *--digest* option.

The WildFly HTTP Management API adheres to the REST principles so the GET operations must be idempotent.

This means that using a request with method **GET** can be used to read the model but **you won't be able to change it**.

You must use **POST** to change the model or read it. A **POST** request may contain the operation either in DMR or in JSON format as its body.

You have to define the **Content-Type=application/json** header in the request to specify that you are using some JSON.

If you want to submit DMR in the request body then the **Content-Type** or the **Accept** header should be **"application/dmr-encoded"**.

GET for Reading

While you can do everything with **POST**, some operations can be called through a 'classical' **GET** request.

These are the supported operations for a GET :

- attribute : for a read-attribute operation
- resource : for a read-resource operation
- resource-description : for a read-resource-description operation
- snapshots : for the list-snapshots operation
- operation-description : for a read-operation-description operation
- operation-names : for ad read-operation-names operation

The URL format is the following one : *http://server:9990/management/<path_to_resource>?operation=<operation_name>&operation_parameter=<value>...*

path_to_resource is the path to the wanted resource replacing all '=' with '/' : thus for example subsystem=undertow/server=default-server becomes subsystem/undertow/server/default-server.

So to read the server-state :

```
http://localhost:9990/management?operation=attribute&name=server-state&json.pretty=1
```

Let's read some resource

- This is simple operation that is equivalent of running :read-attribute(name=server-state) with CLI in root directory
    - Using GET

```
http://localhost:9990/management?operation=attribute&name=server-state&json.pretty=1
```

    - Using POST

```
$ curl --digest -L -D - http://localhost:9990/management --header "Content-Type:
application/json" -d
'{"operation":"read-attribute","name":"server-state","json.pretty":1}' -u admin
Enter host password for user 'admin':
HTTP/1.1 401 Unauthorized
Connection: keep-alive
WWW-Authenticate: Digest
realm="ManagementRealm",domain="/management",nonce="P80WU3BANtQNMTQwNjg5Mzc5MDQ2MlpjmR
77
Content-Type: text/html
Date: Fri, 01 Aug 2014 11:49:50 GMT

HTTP/1.1 200 OK
Connection: keep-alive
Authentication-Info:
nextnonce="M+h9aADejeINMTQwNjg5Mzc5MDQ2OPQbHKdAS8pRE8BbGEDY5uI="
Content-Type: application/json; charset=utf-8
Content-Length: 55
Date: Fri, 01 Aug 2014 11:49:50 GMT

{
    "outcome" : "success",
    "result" : "running"
}
```

- Here's an example of an operation on a resource with a nested address and passed parameters. This is same as if you would run **/host=master/server=server-01:read-attribute(name=server-state)**

```
$ curl --digest -L -D - http://localhost:9990/management --header "Content-Type:
application/json" -d
'{"operation":"read-attribute","address":[{"host":"master"},{"server":"server-01"}],"name":"server
200 OK
Transfer-encoding: chunked
Content-type: application/json
Date: Tue, 17 Apr 2012 04:02:24 GMT

{
    "outcome" : "success",
    "result" : "running"
}
```

- Following example will get us information from http connection in undertow subsystem including run-time attributes
  This is the same as running /subsystem=undertow/server=default-server:read-resource(include-runtime=true,recursive=true) in CLI
  - Using GET

```
http://localhost:9990/management/subsystem/undertow/server/default-server?operation=re
"default-host" : "default-host",
    "servlet-container" : "default",
    "ajp-listener" : null,
    "host" : {"default-host" : {
        "alias" : ["localhost"],
        "default-web-module" : "ROOT.war",
        "filter-ref" : {
            "server-header" : {"predicate" : null},
            "x-powered-by-header" : {"predicate" : null}
        },
        "location" : {"/" : {
            "handler" : "welcome-content",
            "filter-ref" : null
        }},
        "setting" : null
    }},
    "http-listener" : {"default" : {
        "allow-encoded-slash" : false,
        "allow-equals-in-cookie-value" : false,
        "always-set-keep-alive" : true,
        "buffer-pipelined-data" : true,
        "buffer-pool" : "default",
        "certificate-forwarding" : false,
        "decode-url" : true,
        "enabled" : true,
        "max-buffered-request-size" : 16384,
        "max-cookies" : 200,
        "max-header-size" : 51200,
        "max-headers" : 200,
        "max-parameters" : 1000,
        "max-post-size" : 10485760,
        "proxy-address-forwarding" : false,
        "read-timeout" : null,
        "receive-buffer" : null,
        "record-request-start-time" : false,
        "redirect-socket" : "https",
        "send-buffer" : null,
        "socket-binding" : "http",
        "tcp-backlog" : null,
        "tcp-keep-alive" : null,
        "url-charset" : "UTF-8",
        "worker" : "default",
        "write-timeout" : null
    }},
    "https-listener" : null
}
```

- Using POST

```
$ curl --digest -D - http://localhost:9990/management --header "Content-Type:
application/json" -d '{"operation":"read-resource", "include-runtime":"true" ,
"recursive":"true", "address":["subsystem","undertow","server","default-server"],
"json.pretty":1}' -u admin:admin
HTTP/1.1 401 Unauthorized
Connection: keep-alive
WWW-Authenticate: Digest
realm="ManagementRealm",domain="/management",nonce="a3paQ9E0/l8NMTQwNjg5OTU0NDk4OKjmim
77
Content-Type: text/html
Date: Fri, 01 Aug 2014 13:25:44 GMT

HTTP/1.1 200 OK
Connection: keep-alive
Authentication-Info:
nextnonce="nTOSJd3ufO4NMTQwNjg5OTU0NDk5MeUsRw5rKXUT4Qvk1nbrG5c="
Content-Type: application/json; charset=utf-8
Content-Length: 1729
Date: Fri, 01 Aug 2014 13:25:45 GMT

{
    "outcome" : "success",
    "result" : {
        "default-host" : "default-host",
        "servlet-container" : "default",
        "ajp-listener" : null,
        "host" : {"default-host" : {
            "alias" : ["localhost"],
            "default-web-module" : "ROOT.war",
            "filter-ref" : {
                "server-header" : {"predicate" : null},
                "x-powered-by-header" : {"predicate" : null}
            },
            "location" : {"/" : {
                "handler" : "welcome-content",
                "filter-ref" : null
            }},
            "setting" : null
        }},
        "http-listener" : {"default" : {
            "allow-encoded-slash" : false,
            "allow-equals-in-cookie-value" : false,
            "always-set-keep-alive" : true,
            "buffer-pipelined-data" : true,
            "buffer-pool" : "default",
            "certificate-forwarding" : false,
            "decode-url" : true,
            "enabled" : true,
            "max-buffered-request-size" : 16384,
            "max-cookies" : 200,
            "max-header-size" : 51200,
            "max-headers" : 200,
            "max-parameters" : 1000,
            "max-post-size" : 10485760,
```

```
            "proxy-address-forwarding" : false,
            "read-timeout" : null,
            "receive-buffer" : null,
            "record-request-start-time" : false,
            "redirect-socket" : "https",
            "send-buffer" : null,
            "socket-binding" : "http",
            "tcp-backlog" : null,
            "tcp-keep-alive" : null,
            "url-charset" : "UTF-8",
            "worker" : "default",
            "write-timeout" : null
        }},
        "https-listener" : null
    }
}
```

- You may also used some encoded DMR but the result won't be human readable

```
curl --digest -u admin:admin --header "Content-Type: application/dmr-encoded" -d
bwAAAAMACW9wZXJhdGlvbnMADXJlYWQtcmVzb3VyY2UAB2FkZHJlc3NsAAAAAAAHcmVjdXJzZVoB
http://localhost:9990/management
```

- You can deploy applications on the server
  - First upload the file which will create a managed content. You will have to use http://localhost:9990/management/*add-content*

```
curl --digest -u admin:admin --form file=@tiny-webapp.war
http://localhost:9990/management/add-content
{"outcome" : "success", "result" : { "BYTES_VALUE" : "+QJlHTDrogO9pm/57GkT/vxWNz0="
}}
```

  - Now let's deploy the application

```
curl --digest -u admin:admin -L --header "Content-Type: application/json" -d
'{"content":[{"hash": {"BYTES_VALUE" : "+QJlHTDrogO9pm/57GkT/vxWNz0="}}],
"address": [{"deployment":"tiny-webapp.war"}], "operation":"add",
"enabled":"true"}' http://localhost:9990/management
{"outcome" : "success"}
```

Using some JAX-RS code

```
HttpAuthenticationFeature feature = HttpAuthenticationFeature.digest("admin", "admin");
Client client = ClientBuilder.newClient();
client.register(feature);
Entity<SimpleOperation> operation = Entity.entity(
    new SimpleOperation("read-resource", true, "subsystem", "undertow", "server",
"default-server"),
    MediaType.APPLICATION_JSON_TYPE);
WebTarget managementResource = client.target("http://localhost:9990/management");
String response = managementResource.request(MediaType.APPLICATION_JSON_TYPE)
    .header("Content-type", MediaType.APPLICATION_JSON)
    .post(operation, String.class);
System.out.println(response);


{"outcome" : "success", "result" : {"default-host" : "default-host", "servlet-container" :
"default", "ajp-listener" : null, "host" : {"default-host" : {"alias" : ["localhost"],
"default-web-module" : "ROOT.war", "filter-ref" : {"server-header" : {"predicate" : null},
"x-powered-by-header" : {"predicate" : null}}, "location" : {"/" : {"handler" :
"welcome-content", "filter-ref" : null}}, "setting" : null}}, "http-listener" : {"default" :
{"allow-encoded-slash" : false, "allow-equals-in-cookie-value" : false, "always-set-keep-alive"
: true, "buffer-pipelined-data" : true, "buffer-pool" : "default", "certificate-forwarding" :
false, "decode-url" : true, "enabled" : true, "max-buffered-request-size" : 16384, "max-cookies"
: 200, "max-header-size" : 51200, "max-headers" : 200, "max-parameters" : 1000, "max-post-size"
: 10485760, "proxy-address-forwarding" : false, "read-timeout" : null, "receive-buffer" : null,
"record-request-start-time" : false, "redirect-socket" : "https", "send-buffer" : null,
"socket-binding" : "http", "tcp-backlog" : null, "tcp-keep-alive" : null, "url-charset" :
"UTF-8", "worker" : "default", "write-timeout" : null}}, "https-listener" : null}}
```

5.17.9 The native management API

A standalone WildFly process, or a managed domain Domain Controller or slave Host Controller process can be configured to listen for remote management requests using its "native management interface":

```
<native-interface interface="management" port="9999" security-realm="ManagementRealm"/>
```

(See standalone/configuration/standalone.xml or domain/configuration/host.xml)

The CLI tool that comes with the application server uses this interface, and user can develop custom clients that use it as well. In this section we'll cover the basics on how to develop such a client. We'll also cover details on the format of low-level management operation requests and responses – information that should prove useful for users of the CLI tool as well.

Native Management Client Dependencies

The native management interface uses an open protocol based on the JBoss Remoting library. JBoss Remoting is used to establish a communication channel from the client to the process being managed. Once the communication channel is established the primary traffic over the channel is management requests initiated by the client and asynchronous responses from the target process.

A custom Java-based client should have the maven artifact `org.jboss.as:jboss-as-controller-client` and its dependencies on the classpath. The other dependencies are:

Maven Artifact	Purpose
`org.jboss.remoting:jboss-remoting`	Remote communication
`org.jboss:jboss-dmr`	Detyped representation of the management model
`org.jboss.as:jboss-as-protocol`	Wire protocol for remote WildFly management
`org.jboss.sasl:jboss-sasl`	SASL authentication
`org.jboss.xnio:xnio-api`	Non-blocking IO
`org.jboss.xnio:xnio-nio`	Non-blocking IO
`org.jboss.logging:jboss-logging`	Logging
`org.jboss.threads:jboss-threads`	Thread management
`org.jboss.marshalling:jboss-marshalling`	Marshalling and unmarshalling data to/from streams

The client API is entirely within the `org.jboss.as:jboss-as-controller-client` artifact; the other dependencies are part of the internal implementation of `org.jboss.as:jboss-as-controller-client` and are not compile-time dependencies of any custom client based on it.

The management protocol is an open protocol, so a completely custom client could be developed without using these libraries (e.g. using Python or some other language.)

Working with a ModelControllerClient

The `org.jboss.as.controller.client.ModelControllerClient` class is the main class a custom client would use to manage a WildFly server instance or a Domain Controller or slave Host Controller.

The custom client must have maven artifact `org.jboss.as:jboss-as-controller-client` and its dependencies on the classpath.

Creating the ModelControllerClient

To create a management client that can connect to your target process's native management socket, simply:

```
ModelControllerClient client =
ModelControllerClient.Factory.create(InetAddress.getByName("localhost"), 9999);
```

The address and port are what is configured in the target process' `<management><management-interfaces><native-interface.../>` element.

Typically, however, the native management interface will be secured, requiring clients to authenticate. On the client side, the custom client will need to provide the user's authentication credentials, obtained in whatever manner is appropriate for the client (e.g. from a dialog box in a GUI-based client.) Access to these credentials is provided by passing in an implementation of the `javax.security.auth.callback.CallbackHandler` interface. For example:

```
static ModelControllerClient createClient(final InetAddress host, final int port,
                  final String username, final char[] password, final String securityRealmName)
{

    final CallbackHandler callbackHandler = new CallbackHandler() {

        public void handle(Callback[] callbacks) throws IOException,
UnsupportedCallbackException {
            for (Callback current : callbacks) {
                if (current instanceof NameCallback) {
                    NameCallback ncb = (NameCallback) current;
                    ncb.setName(username);
                } else if (current instanceof PasswordCallback) {
                    PasswordCallback pcb = (PasswordCallback) current;
                    pcb.setPassword(password.toCharArray());
                } else if (current instanceof RealmCallback) {
                    RealmCallback rcb = (RealmCallback) current;
                    rcb.setText(rcb.getDefaultText());
                } else {
                    throw new UnsupportedCallbackException(current);
                }
            }
        }
    };

    return ModelControllerClient.Factory.create(host, port, callbackHandler);
}
```

Creating an operation request object

Management requests are formulated using the `org.jboss.dmr.ModelNode` class from the `jboss-dmr` library. The `jboss-dmr` library allows the complete WildFly management model to be expressed using a very small number of Java types. See Detyped management and the jboss-dmr library for full details on using this library.

Let's show an example of creating an operation request object that can be used to read the resource description for the web subsystem's HTTP connector:

```
ModelNode op = new ModelNode();
op.get("operation").set("read-resource-description");

ModelNode address = op.get("address");
address.add("subsystem", "web");
address.add("connector", "http");

op.get("recursive").set(true);
op.get("operations").set(true);
```

What we've done here is created a ModelNode of type `ModelType.OBJECT` with the following fields:

- `operation` – the name of the operation to invoke. All operation requests **must** include this field and its value must be a String.
- `address` – the address of the resource to invoke the operation against. This field's must be of `ModelType.LIST` with each element in the list being a `ModelType.PROPERTY`. If this field is omitted the operation will target the root resource. The operation can be targeted at any address in the management model; here we are targeting it at the resource for the web subsystem's http connector.

In this case, the request includes two optional parameters:

- `recursive` – true means you want the description of child resources under this resource. Default is false
- `operations` – true means you want the description of operations exposed by the resource to be included. Default is false.

Different operations take different parameters, and some take no parameters at all.

See Format of a Detyped Operation Request for full details on the structure of a ModelNode that will represent an operation request.

The example above produces an operation request ModelNode equivalent to what the CLI produces internally when it parses and executes the following low-level CLI command:

```
[localhost:9999 /]
/subsystem=web/connector=http:read-resource-description(recursive=true,operations=true)
```

Execute the operation and manipulate the result:

The `execute` method sends the operation request ModelNode to the process being managed and returns a ModelNode the contains the process' response:

```
ModelNode returnVal = client.execute(op);
System.out.println(returnVal.get("result").toString());
```

See Format of a Detyped Operation Response for general details on the structure of the "returnVal" ModelNode.

The `execute` operation shown above will block the calling thread until the response is received from the process being managed. `ModelControllerClient` also exposes and API allowing asynchronous invocation:

```
Future<ModelNode> future = client.executeAsync(op);
. . .  // do other stuff
ModelNode returnVal = future.get();
System.out.println(returnVal.get("result").toString());
```

Close the ModelControllerClient

A `ModelControllerClient` can be reused for multiple requests. Creating a new `ModelControllerClient` for each request is an anti-pattern. However, when the `ModelControllerClient` is no longer needed, it should always be explicitly closed, allowing it to close down any connections to the process it was managing and release other resources:

```
client.close();
```

Format of a Detyped Operation Request

The basic method a user of the WildFly 8 programmatic management API would use is very simple:

```
ModelNode execute(ModelNode operation) throws IOException;
```

where the return value is the detyped representation of the response, and `operation` is the detyped representation of the operation being invoked.

The purpose of this section is to document the structure of `operation`.

See Format of a Detyped Operation Response for a discussion of the format of the response.

Simple Operations

A text representation of simple operation would look like this:

```
{
    "operation" => "write-attribute",
    "address" => [
        ("profile" => "production"),
        ("subsystem" => "threads"),
        ("bounded-queue-thread-pool" => "pool1")
    ],
    "name" => "count",
    "value" => 20
}
```

Java code to produce that output would be:

```
ModelNode op = new ModelNode();
op.get("operation").set("write-attribute");
ModelNode addr = op.get("address");
addr.add("profile", "production");
addr.add("subsystem", "threads");
addr.add("bounded-queue-thread-pool", "pool1");
op.get("name").set("count");
op.get("value").set(20);
System.out.println(op);
```

The order in which the outermost elements appear in the request is not relevant. The required elements are:

- `operation` – String – The name of the operation being invoked.
- `address` – the address of the managed resource against which the request should be executed. If not set, the address is the root resource. The address is an ordered list of key-value pairs describing where the resource resides in the overall management resource tree. Management resources are organized in a tree, so the order in which elements in the address occur is important.

The other key/value pairs are parameter names and their values. The names and values should match what is specified in the operation's description.

Parameters may have any name, except for the reserved words `operation`, `address` and `operation-headers`.

Operation Headers

Besides the special operation and address values discussed above, operation requests can also include special "header" values that help control how the operation executes. These headers are created under the special reserved word `operation-headers`:

```
ModelNode op = new ModelNode();
op.get("operation").set("write-attribute");
ModelNode addr = op.get("address");
addr.add("base", "domain");
addr.add("profile", "production");
addr.add("subsystem", "threads");
addr.add("bounded-queue-thread-pool", "pool1");
op.get("name").set("count");
op.get("value").set(20);
op.get("operation-headers", "rollback-on-runtime-failure").set(false);
System.out.println(op);
```

This produces:

```
{
    "operation" => "write-attribute",
    "address" => [
        ("profile" => "production"),
        ("subsystem" => "threads"),
        ("bounded-queue-thread-pool" => "pool1")
    ],
    "name" => "count",
    "value" => 20,
    "operation-headers" => {
        "rollback-on-runtime-failure => false
    }
}
```

The following operation headers are supported:

- `rollback-on-runtime-failure` – boolean, optional, defaults to true. Whether an operation that successfully updates the persistent configuration model should be reverted if it fails to apply to the runtime. Operations that affect the persistent configuration are applied in two stages – first to the configuration model and then to the actual running services. If there is an error applying to the configuration model the operation will be aborted with no configuration change and no change to running services will be attempted. However, operations are allowed to change the configuration model even if there is a failure to apply the change to the running services – if and only if this `rollback-on-runtime-failure` header is set to `false`. So, this header only deals with what happens if there is a problem applying an operation to the running state of a server (e.g. actually increasing the size of a runtime thread pool.)
- `rollout-plan` – only relevant to requests made to a Domain Controller or Host Controller. See "Operations with a Rollout Plan" for details.
- `allow-resource-service-restart` – boolean, optional, defaults to false. Whether an operation that requires restarting some runtime services in order to take effect should do so. See discussion of `resource-services` in the "Applying Updates to Runtime Services" section of the Description of the Management Model section for further details.
- `roles` – String or list of strings. Name(s) of RBAC role(s) the permissions for which should be used when making access control decisions instead of those from the roles normally associated with the user invoking the operation. Only respected if the user is normally associated with a role with all permissions (i.e. SuperUser), meaning this can only be used to reduce permissions for a caller, not to increase permissions.
- `blocking-timeout` – int, optional, defaults to 300. Maximum time, in seconds, that the operation should block at various points waiting for completion. If this period is exceeded, the operation will roll back. Does not represent an overall maximum execution time for an operation; rather it is meant to serve as a sort of fail-safe measure to prevent problematic operations indefinitely tying up resources.

Composite Operations

The root resource for a Domain or Host Controller or an individual server will expose an operation named "`composite`". This operation executes a list of other operations as an atomic unit (although the atomicity requirement can be relaxed. The structure of the request for the "`composite`" operation has the same fundamental structure as a simple operation (i.e. operation name, address, params as key value pairs).

```
{
    "operation" => "composite",
    "address" => [],
    "steps" => [
         {
              "operation" => "write-attribute",
              "address" => [
                   ("profile" => "production"),
                   ("subsystem" => "threads"),
                   ("bounded-queue-thread-pool" => "pool1")
              ],
              "count" => "count",
              "value" => 20
         },
         {
              "operation" => "write-attribute",
              "address" => [
                   ("profile" => "production"),
                   ("subsystem" => "threads"),
                   ("bounded-queue-thread-pool" => "pool2")
              ],
              "name" => "count",
              "value" => 10
         }
    ],
    "operation-headers" => {
        "rollback-on-runtime-failure => false
    }
}
```

The "composite" operation takes a single parameter:

- `steps` – a list, where each item in the list has the same structure as a simple operation request. In the example above each of the two steps is modifying the thread pool configuration for a different pool. There need not be any particular relationship between the steps. Note that the `rollback-on-runtime-failure` and `rollout-plan` operation headers are not supported for the individual steps in a composite operation.

The `rollback-on-runtime-failure` operation header discussed above has a particular meaning when applied to a composite operation, controlling whether steps that successfully execute should be reverted if other steps fail at runtime. Note that if any steps modify the persistent configuration, and any of those steps fail, all steps will be reverted. Partial/incomplete changes to the persistent configuration are not allowed.

Operations with a Rollout Plan

Operations targeted at domain or host level resources can potentially impact multiple servers. Such operations can include a "rollout plan" detailing the sequence in which the operation should be applied to servers as well as policies for detailing whether the operation should be reverted if it fails to execute successfully on some servers.

If the operation includes a rollout plan, the structure is as follows:

```
{
    "operation" => "write-attribute",
    "address" => [
        ("profile" => "production"),
        ("subsystem" => "threads"),
        ("bounded-queue-thread-pool" => "pool1")
    ],
    "name" => "count",
    "value" => 20,
    "operation-headers" => {
        "rollout-plan" => {
            "in-series" => [
                {
                    "concurrent-groups" => {
                        "groupA" => {
                            "rolling-to-servers" => true,
                            "max-failure-percentage" => 20
                        },
                        "groupB" => undefined
                    }
                },
                {
                   "server-group" => {
                        "groupC" => {
                            "rolling-to-servers" => false,
                            "max-failed-servers" => 1
                        }
                    }
                },
                {
                    "concurrent-groups" => {
                        "groupD" => {
                            "rolling-to-servers" => true,
                            "max-failure-percentage" => 20
                        },
                        "groupE" => undefined
                    }
                }
            ],
            "rollback-across-groups" => true
        }
    }
}
```

As you can see, the rollout plan is another structure in the operation-headers section. The root node of the structure allows two children:

- `in-series` – a list – A list of activities that are to be performed in series, with each activity reaching completion before the next step is executed. Each activity involves the application of the operation to the servers in one or more server groups. See below for details on each element in the list.
- `rollback-across-groups` – boolean – indicates whether the need to rollback the operation on all the servers in one server group should trigger a rollback across all the server groups. This is an optional setting, and defaults to `false`.

Each element in the list under the `in-series` node must have one or the other of the following structures:

- `concurrent-groups` – a map of server group names to policies controlling how the operation should be applied to that server group. For each server group in the map, the operation may be applied concurrently. See below for details on the per-server-group policy configuration.
- `server-group` – a single key/value mapping of a server group name to a policy controlling how the operation should be applied to that server group. See below for details on the policy configuration. (Note: there is no difference in plan execution between this and a "`concurrent-groups`" map with a single entry.)

The policy controlling how the operation is applied to the servers within a server group has the following elements, each of which is optional:

- `rolling-to-servers` – boolean – If true, the operation will be applied to each server in the group in series. If false or not specified, the operation will be applied to the servers in the group concurrently.
- `max-failed-servers` – int – Maximum number of servers in the group that can fail to apply the operation before it should be reverted on all servers in the group. The default value if not specified is zero; i.e. failure on any server triggers rollback across the group.
- `max-failure-percentage` – int between 0 and 100 – Maximum percentage of the total number of servers in the group that can fail to apply the operation before it should be reverted on all servers in the group. The default value if not specified is zero; i.e. failure on any server triggers rollback across the group.

If both `max-failed-servers` and `max-failure-percentage` are set, `max-failure-percentage` takes precedence.

Looking at the (contrived) example above, application of the operation to the servers in the domain would be done in 3 phases. If the policy for any server group triggers a rollback of the operation across the server group, all other server groups will be rolled back as well. The 3 phases are:

1. Server groups groupA and groupB will have the operation applied concurrently. The operation will be applied to the servers in groupA in series, while all servers in groupB will handle the operation concurrently. If more than 20% of the servers in groupA fail to apply the operation, it will be rolled back across that group. If any servers in groupB fail to apply the operation it will be rolled back across that group.
2. Once all servers in groupA and groupB are complete, the operation will be applied to the servers in groupC. Those servers will handle the operation concurrently. If more than one server in groupC fails to apply the operation it will be rolled back across that group.
3. Once all servers in groupC are complete, server groups groupD and groupE will have the operation applied concurrently. The operation will be applied to the servers in groupD in series, while all servers in groupE will handle the operation concurrently. If more than 20% of the servers in groupD fail to apply the operation, it will be rolled back across that group. If any servers in groupE fail to apply the operation it will be rolled back across that group.

Default Rollout Plan

All operations that impact multiple servers will be executed with a rollout plan. However, actually specifying the rollout plan in the operation request is not required. If no `rollout-plan` operation header is specified, a default plan will be generated. The plan will have the following characteristics:

- There will only be a single high level phase. All server groups affected by the operation will have the operation applied concurrently.
- Within each server group, the operation will be applied to all servers concurrently.
- Failure on any server in a server group will cause rollback across the group.
- Failure of any server group will result in rollback of all other server groups.

Creating and reusing a Rollout Plan

Since a rollout plan may be quite complex, having to pass it as a header every time can become quickly painful. So instead we can store it in the model and then reference it when we want to use it.
To create a rollout plan you can use the operation `rollout-plan add` like this :

```
rollout-plan add --name=simple --content={"rollout-plan" => {"in-series" => [{"server-group" =>
{"main-server-group" => {"rolling-to-servers" => false,"max-failed-servers" => 1}}},
{"server-group" => {"other-server-group" => {"rolling-to-servers" =>
true,"max-failure-percentage" => 20}}}],"rollback-across-groups" => true}}
```

This will create a rollout plan called `simple` in the content repository.

```
[domain@192.168.1.20:9999 /]
/management-client-content=rollout-plans/rollout-plan=simple:read-resource
{
    "outcome" => "success",
    "result" => {
        "content" => {"rollout-plan" => {
            "in-series" => [
                {"server-group" => {"main-server-group" => {
                    "rolling-to-servers" => false,
                    "max-failed-servers" => 1
                }}},
                {"server-group" => {"other-server-group" => {
                    "rolling-to-servers" => true,
                    "max-failure-percentage" => 20
                }}}
            ],
            "rollback-across-groups" => true
        }},
        "hash" => bytes {
            0x13, 0x12, 0x76, 0x65, 0x8a, 0x28, 0xb8, 0xbc,
            0x34, 0x3c, 0xe9, 0xe6, 0x9f, 0x24, 0x05, 0xd2,
            0x30, 0xff, 0xa4, 0x34
        }
    }
}
```

Now you may reference the roolout plan in your command by adding a header just like this :

```
deploy /quickstart/ejb-in-war/target/wildfly-ejb-in-war.war --all-server-groups
--headers={rollout name=simple}
```

Format of a Detyped Operation Response

As noted previously, the basic method a user of the WildFly 8 programmatic management API would use is very simple:

```
ModelNode execute(ModelNode operation) throws IOException;
```

where the return value is the detyped representation of the response, and `operation` is the detyped representation of the operating being invoked.

The purpose of this section is to document the structure of the return value.

For the format of the request, see Format of a Detyped Operation Request.

Simple Responses

Simple responses are provided by the following types of operations:

- Non-composite operations that target a single server. (See below for more on composite operations).
- Non-composite operations that target a Domain Controller or slave Host Controller and don't require the responder to apply the operation on multiple servers and aggregate their results (e.g. a simple read of a domain configuration property.)

The response will always include a simple boolean outcome field, with one of three possible values:

- `success` – the operation executed successfully
- `failed` – the operation failed
- `cancelled` – the execution of the operation was cancelled. (This would be an unusual outcome for a simple operation which would generally very rapidly reach a point in its execution where it couldn't be cancelled.)

The other fields in the response will depend on whether the operation was successful.

The response for a failed operation:

```
{
    "outcome" => "failed",
    "failure-description" => "[JBAS-12345] Some failure message"
}
```

A response for a successful operation will include an additional field:

- `result` – the return value, or `undefined` for void operations or those that return null

A non-void result:

```
{
    "outcome" => "success",
    "result" => {
        "name" => "Brian",
        "age" => 22
    }
}
```

A void result:

```
{
    "outcome" => "success",
    "result" => undefined
}
```

The response for a cancelled operation has no other fields:

```
{
    "outcome" => "cancelled"
}
```

Response Headers

Besides the standard `outcome`, `result` and `failure-description` fields described above, the response may also include various headers that provide more information about the affect of the operation or about the overall state of the server. The headers will be child element under a field named `response-headers`. For example:

```
{
    "outcome" => "success",
    "result" => undefined,
    "response-headers" => {
        "operation-requires-reload" => true,
        "process-state" => "reload-required"
    }
}
```

A response header is typically related to whether an operation could be applied to the targeted runtime without requiring a restart of some or all services, or even of the target process itself. Please see the "Applying Updates to Runtime Services" section of the Description of the Management Model section for a discussion of the basic concepts related to what happens if an operation requires a service restart to be applied.

The current possible response headers are:

- `operation-requires-reload` – boolean – indicates that the specific operation that has generated this response requires a restart of all services in the process in order to take effect in the runtime. This would typically only have a value of 'true'; the absence of the header is the same as a value of 'false.'
- `operation-requires-restart` – boolean – indicates that the specific operation that has generated this response requires a full process restart in order to take effect in the runtime. This would typically only have a value of 'true'; the absence of the header is the same as a value of 'false.'
- `process-state` – enumeration – Provides information about the overall state of the target process. One of the following values:
  - `starting` – the process is starting
  - `running` – the process is in a normal running state. The `process-state` header would typically not be seen with this value; the absence of the header is the same as a value of 'running'.
  - `reload-required` – some operation (not necessarily this one) has executed that requires a restart of all services in order for a configuration change to take effect in the runtime.
  - `restart-required` – some operation (not necessarily this one) has executed that requires a full process restart in order for a configuration change to take effect in the runtime.
  - `stopping` – the process is stopping

Basic Composite Operation Responses

A composite operation is one that incorporates more than one simple operation in a list and executes them atomically. See the "Composite Operations" section for more information.

Basic composite responses are provided by the following types of operations:

- Composite operations that target a single server.
- Composite operations that target a Domain Controller or a slave Host Controller and don't require the responder to apply the operation on multiple servers and aggregate their results (e.g. a list of simple reads of domain configuration properties.)

The high level format of a basic composite operation response is largely the same as that of a simple operation response, although there is an important semantic difference. For a composite operation, the meaning of the outcome flag is controlled by the value of the operation request's `rollback-on-runtime-failure` header field. If that field was `false` (default is true), the outcome flag will be success if all steps were successfully applied to the persistent configuration even if **none** of the composite operation's steps was successfully applied to the runtime.

What's distinctive about a composite operation response is the `result` field. First, even if the operation was not successful, the `result` field will usually be present. (It won't be present if there was some sort of immediate failure that prevented the responder from even attempting to execute the individual operations.) Second, the content of the `result` field will be a map. Each entry in the map will record the result of an element in the `steps` parameter of the composite operation request. The key for each item in the map will be the string "`step-X`" where "X" is the 1-based index of the step's position in the request's `steps` list. So each individual operation in the composite operation will have its result recorded.

The individual operation results will have the same basic format as the simple operation results described above. However, there are some differences from the simple operation case when the individual operation's `outcome` flag is `failed`. These relate to the fact that in a composite operation, individual operations can be rolled back or not even attempted.

If an individual operation was not even attempted (because the overall operation was cancelled or, more likely, a prior operation failed):

```
{
    "outcome" => "cancelled"
}
```

An individual operation that failed and was rolled back:

```
{
    "outcome" => "failed",
    "failure-description" => "[JBAS-12345] Some failure message",
    "rolled-back" => true
}
```

An individual operation that itself succeeded but was rolled back due to failure of another operation:

```
{
    "outcome" => "failed",
    "result" => {
        "name" => "Brian",
        "age" => 22
    },
    "rolled-back" => true
}
```

An operation that failed and was rolled back:

```
{
    "outcome" => "failed",
    "failure-description" => "[JBAS-12345] Some failure message",
    "rolled-back" => true
}
```

Here's an example of the response for a successful 2 step composite operation:

```
{
    "outcome" => "success",
    "result" => [
        {
            "outcome" => "success",
            "result" => {
                "name" => "Brian",
                "age" => 22
            }
        },
        {
            "outcome" => "success",
            "result" => undefined
        }
    ]
}
```

And for a failed 3 step composite operation, where the first step succeeded and the second failed, triggering cancellation of the 3rd and rollback of the others:

```
{
    "outcome" => "failed",
    "failure-description" => "[JBAS-99999] Composite operation failed; see individual operation
results for details",
    "result" => [
        {
            "outcome" => "failed",
            "result" => {
                "name" => "Brian",
                "age" => 22
            },
            "rolled-back" => true
        },
        {
            "outcome" => "failed",
            "failure-description" => "[JBAS-12345] Some failure message",
            "rolled-back" => true
        },
        {
            "outcome" => "cancelled"
        }
    ]
}
```

Multi-Server Responses

Multi-server responses are provided by operations that target a Domain Controller or slave Host Controller and require the responder to apply the operation on multiple servers and aggregate their results (e.g. nearly all domain or host configuration updates.)

Multi-server operations are executed in several stages.

First, the operation may need to be applied against the authoritative configuration model maintained by the Domain Controller (for `domain.xml` confgurations) or a Host Controller (for a `host.xml` configuration). If there is a failure at this stage, the operation is automatically rolled back, with a response like this:

```
{
    "outcome" => "failed",
    "failure-description" => {
        "domain-failure-description" => "[JBAS-33333] Failed to apply X to the domain model"
    }
}
```

If the operation was addressed to the domain model, in the next stage the Domain Controller will ask each slave Host Controller to apply it to its local copy of the domain model. If any Host Controller fails to do so, the Domain Controller will tell all Host Controllers to revert the change, and it will revert the change locally as well. The response to the client will look like this:

```
{
    "outcome" => "failed",
    "failure-description" => {
        "host-failure-descriptions" => {
            "hostA" => "[DOM-3333] Failed to apply to the domain model",
            "hostB" => "[DOM-3333] Failed to apply to the domain model"
        }
    }
}
```

If the preceding stages succeed, the operation will be pushed to all affected servers. If the operation is successful on all servers, the response will look like this (this example operation has a void response, hence the result for each server is undefined):

```
{
    "outcome" => "success",
    "result" => undefined,
    "server-groups" => {
        "groupA" => {
            "serverA-1" => {
                "host" => "host1",
                "response" => {
                    "outcome" => "success",
                    "result" => undefined
                }
            },
            "serverA-2" => {
                "host" => "host2",
                "response" => {
                    "outcome" => "success",
                    "result" => undefined
                }
            }
        },
        "groupB" => {
            "serverB-1" => {
                "host" => "host1",
                "response" => {
                    "outcome" => "success",
                    "result" => undefined
                }
            },
            "serverB-2" => {
                "host" => "host2",
                "response" => {
                    "outcome" => "success",
                    "result" => undefined
                }
            }
        }
    }
}
```

The operation need not succeed on all servers in order to get an `"outcome" => "success"` result. All that is required is that it succeed on at least one server without the rollback policies in the rollout plan triggering a rollback on that server. An example response in such a situation would look like this:

```
{
    "outcome" => "success",
    "result" => undefined,
    "server-groups" => {
        "groupA" => {
            "serverA-1" => {
                "host" => "host1",
                "response" => {
                    "outcome" => "success",
                    "result" => undefined
                }
            },
            "serverA-2" => {
                "host" => "host2",
                "response" => {
                    "outcome" => "success",
                    "result" => undefined
                }
            }
        },
        "groupB" => {
            "serverB-1" => {
                "host" => "host1",
                "response" => {
                    "outcome" => "success",
                    "result" => undefined,
                    "rolled-back" => true
                }
            },
            "serverB-2" => {
                "host" => "host2",
                "response" => {
                    "outcome" => "success",
                    "result" => undefined,
                    "rolled-back" => true
                }
            },
            "serverB-3" => {
                "host" => "host3",
                "response" => {
                    "outcome" => "failed",
                    "failure-description" => "[DOM-4556] Something didn't work right",
                    "rolled-back" => true
                }
            }
        }
    }
}
```

Finally, if the operation fails or is rolled back on all servers, an example response would look like this:

```
{
    "outcome" => "failed",
    "server-groups" => {
        "groupA" => {
            "serverA-1" => {
                "host" => "host1",
                "response" => {
                    "outcome" => "success",
                    "result" => undefined
                }
            },
            "serverA-2" => {
                "host" => "host2",
                "response" => {
                    "outcome" => "success",
                    "result" => undefined
                }
            }
        },
        "groupB" => {
            "serverB-1" => {
                "host" => "host1",
                "response" => {
                    "outcome" => "failed",
                    "result" => undefined,
                    "rolled-back" => true
                }
            },
            "serverB-2" => {
                "host" => "host2",
                "response" => {
                    "outcome" => "failed",
                    "result" => undefined,
                    "rolled-back" => true
                }
            },
            "serverB-3" => {
                "host" => "host3",
                "response" => {
                    "outcome" => "failed",
                    "failure-description" => "[DOM-4556] Something didn't work right",
                    "rolled-back" => true
                }
            }
        }
    }
}
```

5.18 Management Clients

WildFly offers three different approaches to configure and manage servers: a web interface, a command line client and a set of XML configuration files. Regardless of the approach you choose, the configuration is always synchronized across the different views and finally persisted to the XML files.

5.18.1 Web Management Interface

The web interface is a GWT application that uses the HTTP management API to configure a management domain or standalone server.

HTTP Management Endpoint

The HTTP API endpoint is the entry point for management clients that rely on the HTTP protocol to integrate with the management layer. It uses a JSON encoded protocol and a de-typed, RPC style API to describe and execute management operations against a managed domain or standalone server. It's used by the web console, but offers integration capabilities for a wide range of other clients too.

The HTTP API endpoint is co-located with either the domain controller or a standalone server. By default, it runs on port 9990:

```
<management-interfaces>
 [...]
 <http-interface security-realm="ManagementRealm">
  <socket-binding http="management-http"/>
 </http-interface>
<management-interfaces>
```

(See standalone/configuration/standalone.xml or domain/configuration/host.xml)

The HTTP API Endpoint serves two different contexts. One for executing management operations and another one that allows you to access the web interface:

- Domain API: http://<host>:9990/management
- Web Console: http://<host>:9990/console

Accessing the web console

The web console is served through the same port as the HTTP management API. It can be accessed by pointing your browser to:

- http://<host>:9990/console

Default URL

By default the web interface can be accessed here: http://localhost:9990/console.

Default HTTP Management Interface Security

WildFly is distributed secured by default. The default security mechanism is username / password based making use of HTTP Digest for the authentication process.

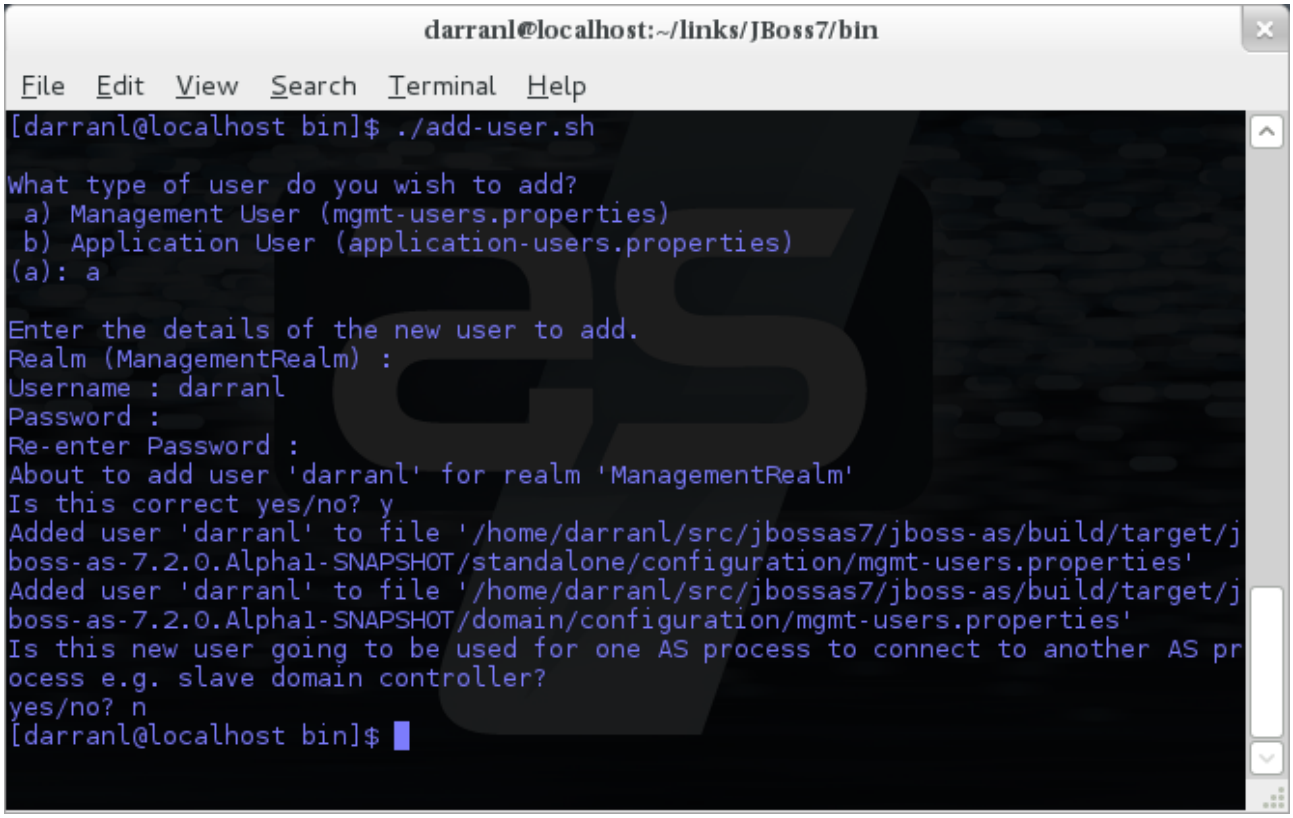
The reason for securing the server by default is so that if the management interfaces are accidentally exposed on a public IP address authentication is required to connect - for this reason there is no default user in the distribution.

If you attempt to connect to the admin console before you have added a user to the server you will be presented with the following screen.

The user are stored in a properties file called mgmt-users.properties under standalone/configuration and domain/configuration depending on the running mode of the server, these files contain the users username along with a pre-prepared hash of the username along with the name of the realm and the users password.

Although the properties files do not contain the plain text passwords they should still be guarded as the pre-prepared hashes could be used to gain access to any server with the same realm if the same user has used the same password.

To manipulate the files and add users we provide a utility add-user.sh and add-user.bat to add the users and generate the hashes, to add a user you should execute the script and follow the guided process.



The full details of the add-user utility are described later but for the purpose of accessing the management interface you need to enter the following values: -

- Type of user - This will be a 'Management User' to selection option a.
- Realm - This MUST match the realm name used in the configuration so unless you have changed the configuration to use a different realm name leave this set as 'ManagementRealm'.
- Username - The username of the user you are adding.
- Password - The users password.

Provided the validation passes you will then be asked to confirm you want to add the user and the properties files will be updated.

For the final question, as this is a user that is going to be accessing the admin console just answer 'n' - this option will be described later for adding slave host controllers that authenticate against a master domain controller but that is a later topic.

Updates to the properties file are picked up in real time so either click 'Try Again' on the error page that was displayed in the browser or navigate to the console again and you should then be prompted to enter the username and password to connect to the server.

5.18.2 Command Line Interface

The Command Line Interface (CLI) is a management tool for a managed domain or standalone server. It allows a user to connect to the domain controller or a standalone server and execute management operations available through the de-typed management model.

Details on how to use the CLI can be found in the Command Line Interface page.

5.18.3 Configuration Files

WildFly stores its configuration in centralized XML configuration files, one per server for standalone servers and, for managed domains, one per host with an additional domain wide policy controlled by the master host. These files are meant to be human-readable and human editable.

The XML configuration files act as a central, authoritative source of configuration. Any configuration changes made via the web interface or the CLI are persisted back to the XML configuration files. If a domain or standalone server is offline, the XML configuration files can be hand edited as well, and any changes will be picked up when the domain or standalone server is next started. However, users are encouraged to use the web interface or the CLI in preference to making offline edits to the configuration files. External changes made to the configuration files while processes are running will not be detected, and may be overwritten.

Standalone Server Configuration File

The XML configuration for a standalone server can be found in the `standalone/configuration` directory. The default configuration file is `standalone/configuration/standalone.xml`.

The `standalone/configuration` directory includes a number of other standard configuration files, e.g. `standalone-full.xml`, `standalone-ha.xml` and `standalone-full-ha.xml` each of which is similar to the default `standalone.xml` file but includes additional subsystems not present in the default configuration. If you prefer to use one of these files as your server configuration, you can specify it with the ~~*c*~~ or *-server-config* command line argument:

- `bin/standalone.sh -c=standalone-full.xml`
- `bin/standalone.sh --server-config=standalone-ha.xml`

Managed Domain Configuration Files

In a managed domain, the XML files are found in the `domain/configuration` directory. There are two types of configuration files – one per host, and then a single domain-wide file managed by the master host, aka the Domain Controller. (For more on the types of processes in a managed domain, see Operating modes.)

Host Specific Configuration – host.xml

When you start a managed domain process, a Host Controller instance is launched, and it parses its own configuration file to determine its own configuration, how it should integrate with the rest of the domain, any host-specific values for settings in the domain wide configuration (e.g. IP addresses) and what servers it should launch. This information is contained in the host-specific configuration file, the default version of which is `domain/configuration/host.xml`.

Each host will have its own variant `host.xml`, with settings appropriate for its role in the domain. WildFly ships with three standard variants:

host-master.xml	A configuration that specifies the Host Controller should become the master, aka the Domain Controller. No servers will be started by this Host Controller, which is a recommended setup for a production master.
host-slave.xml	A configuration that specifies the Host Controller should not become master and instead should register with a remote master and be controlled by it. This configuration launches servers, although a user will likely wish to modify how many servers are launched and what server groups they belong to.
host.xml	The default host configuration, tailored for an easy out of the box experience experimenting with a managed domain. This configuration specifies the Host Controller should become the master, aka the Domain Controller, but it also launches a couple of servers.

Which host-specific configuration should be used can be controlled via the _*--host-config*_ command line argument:

```
$ bin/domain.sh --host-config=host-master.xml
```

Domain Wide Configuration – domain.xml

Once a Host Controller has processed its host-specific configuration, it knows whether it is configured to act as the master Domain Controller. If it is, it must parse the domain wide configuration file, by default located at `domain/configuration/domain.xml`. This file contains the bulk of the settings that should be applied to the servers in the domain when they are launched – among other things, what subsystems they should run with what settings, what sockets should be used, and what deployments should be deployed.

Which domain-wide configuration should be used can be controlled via the _*--domain-config*_ command line argument:

```
$ bin/domain.sh --domain-config=domain-production.xml
```

That argument is only relevant for hosts configured to act as the master.

A slave Host Controller does not usually parse the domain wide configuration file. A slave gets the domain wide configuration from the remote master Domain Controller when it registers with it. A slave also will not persist changes to a `domain.xml` file if one is present on the filesystem. For that reason it is recommended that no `domain.xml` be kept on the filesystem of hosts that will only run as slaves.

A slave can be configured to keep a locally persisted copy of the domain wide configuration and then use it on boot (in case the master is not available.) See *--backup and --cached-dc* under Command line parameters.

5.18.4 Command Line Interface

The Command Line Interface (CLI) is a management tool for a managed domain or standalone server. It allows a user to connect to the domain controller or a standalone server and execute management operations available through the de-typed management model.

Running the CLI

Depending on the operating system, the CLI is launched using `jboss-cli.sh` or `jboss-cli.bat` located in the WildFly `bin` directory. For further information on the default directory structure, please consult the " Getting Started Guide".

The first thing to do after the CLI has started is to connect to a managed WildFly instance. This is done using the command *connect*, e.g.

```
./bin/jboss-cli.sh
You are disconnected at the moment. Type 'connect' to connect to the server
or 'help' for the list of supported commands.
[disconnected /]

[disconnected /] connect
[domain@localhost:9990 /]

[domain@localhost:9990 /] quit
Closed connection to localhost:9990
```

`localhost:9990` is the default host and port combination for the WildFly CLI client.

The host and the port of the server can be provided as an optional parameter, if the server is not listening on localhost:9990.

```
./bin/jboss-cli.sh
You are disconnected at the moment. Type 'connect' to connect to the server
[disconnected /] connect 192.168.0.10:9990
Connected to standalone controller at 192.168.0.1:9990
```

The :9990 is not required as the CLI will use port 9990 by default. The port needs to be provided if the server is listening on some other port.

To terminate the session type *quit*.

The jboss-cli script accepts a --connect parameter: ./jboss-cli.sh --connect

The --controller parameter can be used to specify the host and port of the server: ./jboss-cli.sh --connect --controller=192.168.0.1:9990

Help is also available:

```
[domain@localhost:9990 /] help --commands
Commands available in the current context:
batch               connection-factory  deployment-overlay  if
patch               reload              try
cd                  connection-info     echo                jdbc-driver-info
pwd                 rollout-plan        undeploy
clear               data-source         echo-dmr            jms-queue
quit                run-batch           unset
command             deploy              help                jms-topic
read-attribute      set                 version
connect             deployment-info     history             ls
read-operation      shutdown            xa-data-source
To read a description of a specific command execute 'command_name --help'.
```

interactive Mode

The CLI can also be run in non-interactive mode to support scripts and other types of command line or batch processing. The --command and --commands arguments can be used to pass a command or a list of commands to execute. Additionally a --file argument is supported which enables CLI commands to be provided from a text file.

For example the following command can be used to list all the current deployments

```
$ ./bin/jboss-cli.sh --connect --commands=ls\ deployment
sample.war
osgi-bundle.jar
```

The output can be combined with other shell commands for further processing, for example to find out what .war files are deployed:

```
$ ./bin/jboss-cli.sh --connect --commands=ls\ deployment | grep war
sample.war
```

In order to match a command with its output, you can provide the option --echo-command (or add the XML element <echo-command> to the CLI configuration file) in order to make the CLI to include the prompt + command + options in the output. With this option enabled, any executed command will be added to the output.

Command timeout

By default CLI command and operation executions are not timely bounded. It means that a command never ending its execution will make the CLI process to be stuck and unresponsive. To protect the CLI from this behavior, one can set a command execution timeout.

Command Timeout behavior

In interactive mode, when a timeout occurs, an error message is displayed then the console prompt is made available to type new commands. In non interactive mode (executing a script or a list of commands), when a timeout occurs, an exception is thrown and the CLI execution is stopped. In both modes (interactive and non interactive), when a timeout occurs, the CLI will make a best effort to cancel the associated server side activities.

Configuring the Command timeout

- Add the XML element <*command-timeout*>*{num seconds}*</*command-timeout*> to the CLI XML configuration file.
- Add the option --*command-timeout={num seconds}* to the CLI command line. This will override any value set in the XML configuration file.

Managing the Command Timeout

Once the CLI is running, the timeout can be adjusted to cope with the commands to execute. For example a batch command will need a longer timeout than a non batch one. The command *command-timeout* allows to get, set and reset the command timeout.

Retrieving the command timeout

The command *command-timeout get* displays the current timeout in seconds. A timeout of 0 means no timeout.

```
[standalone@localhost:9990 /] command-timeout get
0
```

Setting the command timeout

The command *command-timeout set* update the timeout value to a number of seconds. If a timeout has been set via configuration (XML file or option), it is overridden by the *set* action.

```
[standalone@localhost:9990 /] command-timeout set 10
```

Resetting the command timeout

The command *command-timeout reset {config|default}* allows to set the timeout to its configuration value (XML file or option) or default value (0 second). If no configuration value is set, resetting to the configuration value sets the timeout to its default value (0 seconds).

```
[standalone@localhost:9990 /] command-timeout reset config
[standalone@localhost:9990 /] command-timeout reset default
```

Default Native Management Interface Security

The native interface shares the same security configuration as the http interface, however we also support a local authentication mechanism which means that the CLI can authenticate against the local WildFly instance without prompting the user for a username and password. This mechanism only works if the user running the CLI has read access to the standalone/tmp/auth folder or domain/tmp/auth folder under the respective WildFly installation - if the local mechanism fails then the CLI will fallback to prompting for a username and password for a user configured as in Default HTTP Interface Security.

Establishing a CLI connection to a remote server will require a username and password by default.

Operation Requests

Operation requests allow for low level interaction with the management model. They are different from the high level commands (i.e. *create-jms-queue*) in that they allow you to read and modify the server configuration as if you were editing the XML configuration files directly. The configuration is represented as a tree of addressable resources, where each node in the tree (aka resource) offers a set of operations to execute.

An operation request basically consists of three parts: The *address*, an *operation name* and an optional set of *parameters*.

The formal specification for an operation request is:

```
[/node-type=node-name (/node-type=node-name)*] : operation-name [(
[parameter-name=parameter-value (,parameter-name=parameter-value)*] )]
```

For example:

```
/subsystem=logging/root-logger=ROOT:change-root-log-level(level=WARN)
```

Tab Completion

Tab-completion is supported for all commands and options, i.e. node-types and node-names, operation names and parameter names. We are also considering adding aliases that are less verbose for the user, and will translate into the corresponding operation requests in the background.

Whitespaces between the separators in the operation request strings are not significant.

Addressing resources

Operation requests might not always have the address part or the parameters. E.g.

```
:read-resource
```

which will list all the node types for the current node.

To syntactically disambiguate between the commands and operations, operations require one of the following prefixes:

To execute an operation against the current node, e.g.

```
cd subsystem=logging
:read-resource(recursive="true")
```

To execute an operation against a child node of the current node, e.g.

```
cd subsystem=logging
./root-logger=ROOT:change-root-log-level(level=WARN)
```

To execute an operation against the root node, e.g.

```
/:read-resource
```

Available Operation Types and Descriptions

The operation types can be distinguished between common operations that exist on any node and specific operations that belong to a particular configuration resource (i.e. subsystem). The common operations are:

- add
- read-attribute
- read-children-names
- read-children-resources
- read-children-types
- read-operation-description
- read-operation-names
- read-resource
- read-resource-description
- remove
- validate-address
- write-attribute

For a list of specific operations (e.g. operations that relate to the logging subsystem) you can always query the model itself. For example, to read the operations supported by the logging subsystem resource on a standalone server:

```
[[standalone@localhost:9990 /] /subsystem=logging:read-operation-names
{
   "outcome" => "success",
   "result" => [
       "add",
       "change-root-log-level",
       "read-attribute",
       "read-children-names",
       "read-children-resources",
       "read-children-types",
       "read-operation-description",
       "read-operation-names",
       "read-resource",
       "read-resource-description",
       "remove-root-logger",
       "root-logger-assign-handler",
       "root-logger-unassign-handler",
       "set-root-logger",
       "validate-address",
       "write-attribute"
   ]
}
```

As you can see, the logging resource offers four additional operations, namely *root-logger-assign-handler*, *root-logger-unassign-handler*, *set-root-logger* and *remove-root-logger*.

Further documentation about a resource or operation can be retrieved through the description:

```
[standalone@localhost:9990 /]
/subsystem=logging:read-operation-description(name=change-root-log-level)
{
   "outcome" => "success",
   "result" => {
       "operation-name" => "change-root-log-level",
       "description" => "Change the root logger level.",
       "request-properties" => {"level" => {
           "type" => STRING,
           "description" => "The log level specifying which message levels will be logged by
this logger.
                            Message levels lower than this value will be discarded.",
           "required" => true
       }}
   }
}
```


Full model

To see the full model enter `:read-resource(recursive=true)`.

Command History

Command (and operation request) history is enabled by default. The history is kept both in-memory and in a file on the disk, i.e. it is preserved between command line sessions. The history file name is .jboss-cli-history and is automatically created in the user's home directory. When the command line interface is launched this file is read and the in-memory history is initialized with its content.

While in the command line session, you can use the arrow keys to go back and forth in the history of commands and operations.

To manipulate the history you can use the *history* command. If executed without any arguments, it will print all the recorded commands and operations (up to the configured maximum, which defaults to 500) from the in-memory history.

history supports three optional arguments:

- *disable* - will disable history expansion (but will not clear the previously recorded history);
- *enabled* - will re-enable history expansion (starting from the last recorded command before the history expansion was disabled);
- *clear* - will clear the in-memory history (but not the file one).

Batch Processing

The batch mode allows one to group commands and operations and execute them together as an atomic unit. If at least one of the commands or operations fails, all the other successfully executed commands and operations in the batch are rolled back.

Not all of the commands are allowed in the batch. For example, commands like *cd*, *ls*, *help*, etc. are not allowed in the batch since they don't translate into operation requests. Only the commands that translate into operation requests are allowed in the batch. The batch, actually, is executed as a composite operation request.

The batch mode is entered by executing command *batch*.

```
[standalone@localhost:9990 /] batch
[standalone@localhost:9990 / #] /subsystem=datasources/data-source="java\:\/H2DS":enable
[standalone@localhost:9990 / #]
/subsystem=messaging-activemq/server=default/jms-queue=newQueue:add
```

You can execute a batch using the *run-batch* command:

```
[standalone@localhost:9990 / #] run-batch
The batch executed successfully.
```

Exit the batch edit mode without losing your changes:

```
[standalone@localhost:9990 / #] holdback-batch
[standalone@localhost:9990 /]
```

Then activate it later on again:

```
[standalone@localhost:9990 /] batch
Re-activated batch
#1 /subsystem=datasources/data-source=java:/H2DS:\/H2DS:enable
```

There are several other notable batch commands available as well (tab complete to see the list):

- *clear-batch*
- *edit-batch-line* (e.g. *edit-batch line 3 create-jms-topic name=mytopic*)
- *remove-batch-line* (e.g. *remove-batch-line 3*)
- *move-batch-line* (e.g. *move-batch-line 3 1*)
- *discard-batch*

5.18.5 Default HTTP Interface Security

WildFly is distributed secured by default. The default security mechanism is username / password based making use of HTTP Digest for the authentication process.

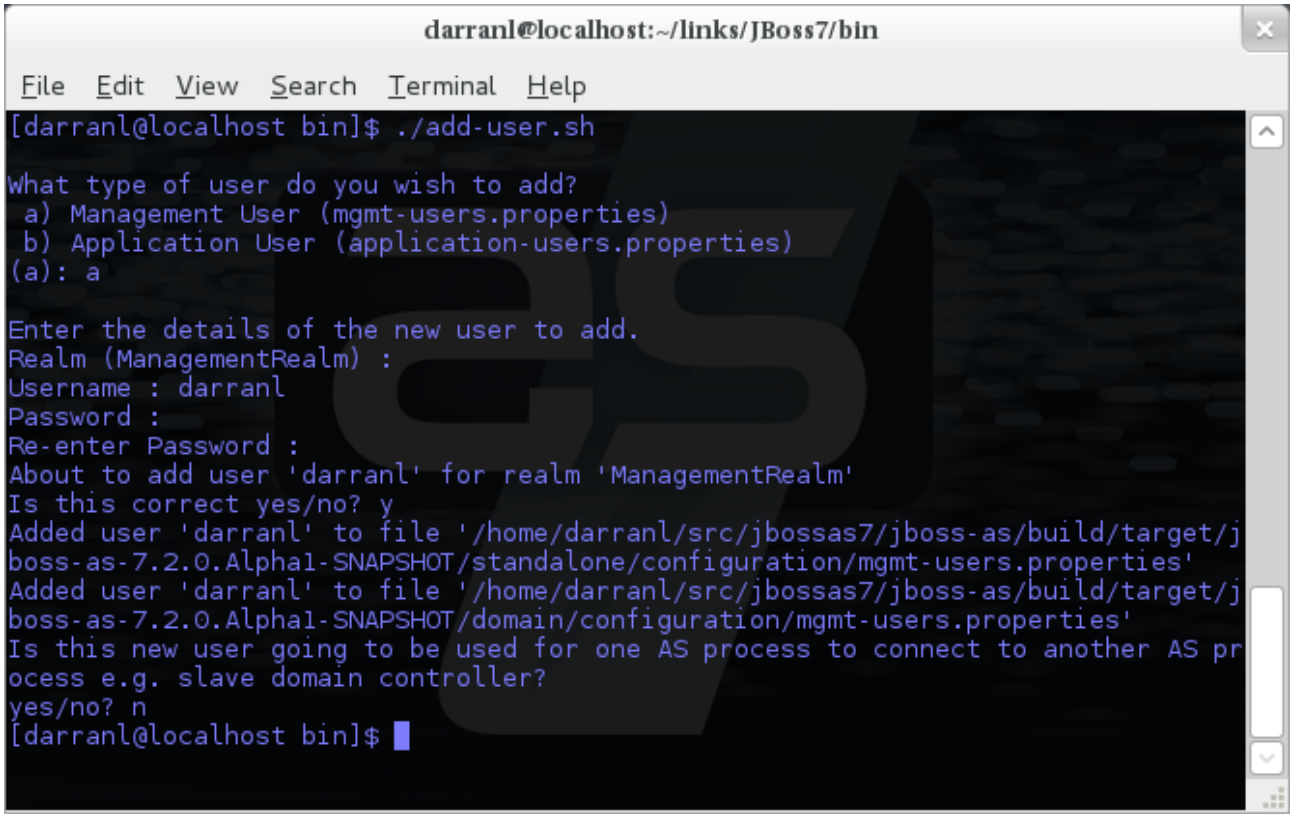
The reason for securing the server by default is so that if the management interfaces are accidentally exposed on a public IP address authentication is required to connect - for this reason there is no default user in the distribution.

If you attempt to connect to the admin console before you have added a user to the server you will be presented with the following screen.

The user are stored in a properties file called mgmt-users.properties under standalone/configuration and domain/configuration depending on the running mode of the server, these files contain the users username along with a pre-prepared hash of the username along with the name of the realm and the users password.

> ⚠ Although the properties files do not contain the plain text passwords they should still be guarded as the pre-prepared hashes could be used to gain access to any server with the same realm if the same user has used the same password.

To manipulate the files and add users we provide a utility add-user.sh and add-user.bat to add the users and generate the hashes, to add a user you should execute the script and follow the guided process.



The full details of the add-user utility are described later but for the purpose of accessing the management interface you need to enter the following values: -

- Type of user - This will be a 'Management User' to selection option a.
- Realm - This MUST match the realm name used in the configuration so unless you have changed the configuration to use a different realm name leave this set as 'ManagementRealm'.
- Username - The username of the user you are adding.
- Password - The users password.

Provided the validation passes you will then be asked to confirm you want to add the user and the properties files will be updated.

For the final question, as this is a user that is going to be accessing the admin console just answer 'n' - this option will be described later for adding slave host controllers that authenticate against a master domain controller but that is a later topic.

Updates to the properties file are picked up in real time so either click 'Try Again' on the error page that was displayed in the browser or navigate to the console again and you should then be prompted to enter the username and password to connect to the server.

5.18.6 Default Native Interface Security

The native interface shares the same security configuration as the http interface, however we also support a local authentication mechanism which means that the CLI can authenticate against the local WildFly instance without prompting the user for a username and password. This mechanism only works if the user running the CLI has read access to the standalone/tmp/auth folder or domain/tmp/auth folder under the respective WildFly installation - if the local mechanism fails then the CLI will fallback to prompting for a username and password for a user configured as in Default HTTP Interface Security.

Establishing a CLI connection to a remote server will require a username and password by default.

5.19 Management tasks

5.19.1 Controlling operation via command line parameters

To start up a WildFly managed domain, execute the `$JBOSS_HOME/bin/domain.sh` script. To start up a standalone server, execute the `$JBOSS_HOME/bin/standalone.sh`. With no arguments, the default configuration is used. You can override the default configuration by providing arguments on the command line, or in your calling script.

System properties

To set a system property, pass its new value using the standard jvm `-Dkey=value` options:

```
$JBOSS_HOME/bin/standalone.sh -Djboss.home.dir=some/location/wildFly \
    -Djboss.server.config.dir=some/location/wildFly/custom-standalone
```

This command starts up a standalone server instance using a non-standard AS home directory and a custom configuration directory. For specific information about system properties, refer to the definitions below.

Instead of passing the parameters directly, you can put them into a properties file, and pass the properties file to the script, as in the two examples below.

```
$JBOSS_HOME/bin/domain.sh --properties=/some/location/jboss.properties
$JBOSS_HOME/bin/domain.sh -P=/some/location/jboss.properties
```

Note however, that properties set this way are not processed as part of JVM launch. They are processed early in the boot process, but this mechanism should not be used for setting properties that control JVM behavior (e.g. java.net.perferIPv4Stack) or the behavior of the JBoss Modules classloading system.

The syntax for passing in parameters and properties files is the same regardless of whether you are running the `domain.sh`, `standalone.sh`, or the Microsoft Windows scripts`domain.bat` or `standalone.bat`.

The properties file is a standard Java property file containing `key=value` pairs:

```
jboss.home.dir=/some/location/wildFly
jboss.domain.config.dir=/some/location/wildFly/custom-domain
```

System properties can also be set via the xml configuration files. Note however that for a standalone server properties set this way will not be set until the xml configuration is parsed and the commands created by the parser have been executed. So this mechanism should not be used for setting properties whose value needs to be set before this point.

Controlling filesystem locations with system properties

The standalone and the managed domain modes each use a default configuration which expects various files and writable directories to exist in standard locations. Each of these standard locations is associated with a system property, which has a default value. To override a system property, pass its new value using the one of the mechanisms above. The locations which can be controlled via system property are:

Standalone

Property name	Usage	Default value
`java.ext.dirs`	The JDK extension directory paths	`null`
`jboss.home.dir`	The root directory of the WildFly installation.	Set by `standalone.sh` to `$JBOSS_HOME`
`jboss.server.base.dir`	The base directory for server content.	`jboss.home.dir`/standalone
`jboss.server.config.dir`	The base configuration directory.	`jboss.server.base.dir` /configuration
`jboss.server.data.dir`	The directory used for persistent data file storage.	`jboss.server.base.dir` /data
`jboss.server.log.dir`	The directory containing the `server.log` file.	`jboss.server.base.dir`/log
`jboss.server.temp.dir`	The directory used for temporary file storage.	`jboss.server.base.dir`/tmp
`jboss.server.deploy.dir`	The directory used to store deployed content	`jboss.server.data.dir` /content

Managed Domain

Property name	Usage	Default value
`jboss.home.dir`	The root directory of the WildFly installation.	Set by `domain.sh` to `$JBOSS_HOME`
`jboss.domain.base.dir`	The base directory for domain content.	`jboss.home.dir`/domain
`jboss.domain.config.dir`	The base configuration directory	`jboss.domain.base.dir`/configuration
`jboss.domain.data.dir`	The directory used for persistent data file storage.	`jboss.domain.base.dir`/data
`jboss.domain.log.dir`	The directory containing the `host-controller.log` and `process-controller.log` files	`jboss.domain.base.dir`/log
`jboss.domain.temp.dir`	The directory used for temporary file storage	`jboss.domain.base.dir`/tmp
`jboss.domain.deployment.dir`	The directory used to store deployed content	`jboss.domain.base.dir`/content
`jboss.domain.servers.dir`	The directory containing the output for the managed server instances	`jboss.domain.base.dir`/servers

Other command line parameters

The first acceptable format for command line arguments to the WildFly launch scripts is

```
--name=value
```

For example:

```
$JBOSS_HOME/bin/standalone.sh --server-config=standalone-ha.xml
```

If the parameter name is a single character, it is prefixed by a single '-' instead of two. Some parameters have both a long and short option.

```
-x=value
```

For example:

```
$JBOSS_HOME/bin/standalone.sh -P=/some/location/jboss.properties
```

For some command line arguments frequently used in previous major releases of WildFly, replacing the "=" in the above examples with a space is supported, for compatibility.

```
-b 192.168.100.10
```

If possible, use the `-x=value` syntax. New parameters will always support this syntax.

The sections below describe the command line parameter names that are available in standalone and domain mode.

Standalone

Name	Default if absent	Value
`--admin-only`	-	Set the server's running type to ADMIN_ONLY causing it to open administrative interfaces and accept management requests but not start other runtime services or accept end user requests.
`--server-config` `-c`	standalone.xml	A relative path which is interpreted to be relative to `jboss.server.config.dir`. The name of the configuration file to use.
`--read-only-server-config`	-	A relative path which is interpreted to be relative to `jboss.server.config.dir`. This is similar to `--server-config` but if this alternative is specified the server will not overwrite the file when the management model is changed. However a full versioned history is maintained of the file.

Managed Domain

Name	Default if absent	Value
`--admin-only`	-	Set the server's running type to ADMIN_ONLY causing it to open administrative interfaces and accept management requests but not start servers or, if this host controller is the master for the domain, accept incoming connections from slave host controllers.
`--domain-config` `-c`	domain.xml	A relative path which is interpreted to be relative to `jboss.domain.config.dir`. The name of the domain wide configuration file to use.
`--read-only-domain-config`	-	A relative path which is interpreted to be relative to `jboss.domain.config.dir`. This is similar to `--domain-config` but if this alternative is specified the host controller will not overwrite the file when the management model is changed. However a full versioned history is maintained of the file.
`--host-config`	host.xml	A relative path which is interpreted to be relative to `jboss.domain.config.dir`. The name of the host-specific configuration file to use.
`--read-only-host-config`	-	A relative path which is interpreted to be relative to `jboss.domain.config.dir`. This is similar to `--host-config` but if this alternative is specified the host controller will not overwrite the file when the management model is changed. However a full versioned history is maintained of the file.

The following parameters take no value and are only usable on slave host controllers (i.e. hosts configured to connect to a `remote` domain controller.)

Name	Function
`--backup`	Causes the slave host controller to create and maintain a local copy (domain.cached-remote.xml) of the domain configuration. If *ignore-unused-configuration* is unset in host.xml, a complete copy of the domain configuration will be stored locally, otherwise the configured value of *ignore-unused-configuration* in host.xml will be used. (See ignore-unused-configuration for more details.)
`--cached-dc`	If the slave host controller is unable to contact the master domain controller to get its configuration at boot, this option will allow the slave host controller to boot and become operational using a previously cached copy of the domain configuration (domain.cached-remote.xml.) If the cached configuration is not present, this boot will fail. This file is created using using one of the following methods: - A previously successful connection to the master domain controller using --backup or --cached-dc. - Copying the domain configuration from an alternative host to domain/configuration/domain.cached-remote.xml. The unavailable master domain controller will be polled periodically for availability, and once becoming available, the slave host controller will reconnect to the master host controller and synchronize the domain configuration. During the interval the master domain controller is unavailable, the slave host controller will not be able make any modifications to the domain configuration, but it may launch servers and handle requests to deployed applications etc.

Common parameters

These parameters apply in both standalone or managed domain mode:

Name	Function
`-b=<value>`	Sets system property `jboss.bind.address` to <value>. See Controlling the Bind Address with -b for further details.
`-b<name>=<value>`	Sets system property `jboss.bind.address.`<name> to <value> where *name* can vary. See Controlling the Bind Address with -b for further details.
`-u=<value>`	Sets system property `jboss.default.multicast.address` to <value>. See Controlling the Default Multicast Address with -u for further details.

`--version`
`-v`
`-V`
	Prints the version of WildFly to standard output and exits the JVM.

`--help` `-h`	Prints a help message explaining the options and exits the JVM.

Controlling the Bind Address with -b

WildFly binds sockets to the IP addresses and interfaces contained in the `<interfaces>` elements in `standalone.xml`, `domain.xml` and `host.xml`. (See Interfaces and Socket Bindings for further information on these elements.) The standard configurations that ship with WildFly includes two interface configurations:

```
<interfaces>
    <interface name="management">
        <inet-address value="${jboss.bind.address.management:127.0.0.1}"/>
    </interface>
    <interface name="public">
       <inet-address value="${jboss.bind.address:127.0.0.1}"/>
    </interface>
</interfaces>
```

Those configurations use the values of system properties `jboss.bind.address.management` and `jboss.bind.address` if they are set. If they are not set, 127.0.0.1 is used for each value.

As noted in Common Parameters, the AS supports the `-b` and `-b<name>` command line switches. The only function of these switches is to set system properties `jboss.bind.address` and `jboss.bind.address.`<name> respectively. However, because of the way the standard WildFly configuration files are set up, using the `-b` switches can indirectly control how the AS binds sockets.

If your interface configurations match those shown above, using this as your launch command causes all sockets associated with interface named "public" to be bound to `192.168.100.10`.

```
$JBOSS_HOME/bin/standalone.sh -b=192.168.100.10
```

In the standard config files, public interfaces are those not associated with server management. Public interfaces handle normal end-user requests.

Interface names

The interface named "public" is not inherently special. It is provided as a convenience. You can name your interfaces to suit your environment.

To bind the public interfaces to all IPv4 addresses (the IPv4 wildcard address), use the following syntax:

```
$JBOSS_HOME/bin/standalone.sh -b=0.0.0.0
```

You can also bind the management interfaces, as follows:

```
$JBOSS_HOME/bin/standalone.sh -bmanagement=192.168.100.10
```

In the standard config files, management interfaces are those sockets associated with server management, such as the socket used by the CLI, the HTTP socket used by the admin console, and the JMX connector socket.

Be Careful

The `-b` switch only controls the interface bindings because the standard config files that ship with WildFly sets things up that way. If you change the `<interfaces>` section in your configuration to no longer use the system properties controlled by `-b`, then setting `-b` in your launch command will have no effect.

For example, this perfectly valid setting for the "public" interface causes `-b` to have no effect on the "public" interface:

```
<interface name="public">
   <nic name="eth0"/>
</interface>
```

The key point is **the contents of the configuration files determine the configuration. Settings like `-b` are not overrides of the configuration files.** They only provide a shorter syntax for setting a system properties that may or may not be referenced in the configuration files. They are provided as a convenience, and you can choose to modify your configuration to ignore them.

Controlling the Default Multicast Address with -u

WildFly may use multicast communication for some services, particularly those involving high availability clustering. The multicast addresses and ports used are configured using the `socket-binding` elements in `standalone.xml` and `domain.xml`. (See Socket Bindings for further information on these elements.) The standard HA configurations that ship with WildFly include two socket binding configurations that use a default multicast address:

```
<socket-binding name="jgroups-mping" port="0"
multicast-address="${jboss.default.multicast.address:230.0.0.4}" multicast-port="45700"/>
<socket-binding name="jgroups-udp" port="55200"
multicast-address="${jboss.default.multicast.address:230.0.0.4}" multicast-port="45688"/>
```

Those configurations use the values of system property `jboss.default.multicast.address` if it is set. If it is not set, 230.0.0.4 is used for each value. (The configuration may include other socket bindings for multicast-based services that are not meant to use the default multicast address; e.g. a binding the mod-cluster services use to communicate on a separate address/port with Apache httpd servers.)

As noted in Common Parameters, the AS supports the `-u` command line switch. The only function of this switch is to set system property `jboss.default.multicast.address`. However, because of the way the standard AS configuration files are set up, using the `-u` switches can indirectly control how the AS uses multicast.

If your socket binding configurations match those shown above, using this as your launch command causes the service using those sockets configurations to be communicate over multicast address `230.0.1.2`.

```
$JBOSS_HOME/bin/standalone.sh -u=230.0.1.2
```

Be Careful

As with the `-b` switch, the `-u` switch only controls the multicast address used because the standard config files that ship with WildFly sets things up that way. If you change the `<socket-binding>` sections in your configuration to no longer use the system properties controlled by `-u`, then setting `-u` in your launch command will have no effect.

5.19.2 Suspend, resume and graceful shutdown

Core Concepts

Wildfly introduces the ability to suspend and resume servers. This can be combined with shutdown to enable the server to gracefully finish processing all active requests and then shut down. When a server is suspended it will immediately stop accepting new requests, but wait for existing request to complete. A suspended server can be resumed at any point, and will begin processing requests immediately. Suspending and resuming has no effect on deployment state (e.g. if a server is suspended singleton EJB's will not be destroyed). As of Wildfly 11 it is also possible to start a server in suspended mode which means it will not accept requests until it has been resumed, servers will also be suspended during the boot process, so no requests will be accepted until the startup process is 100% complete.

Suspend/Resume has no effect on management operations, management operations can still be performed while a server is suspended. If you wish to perform a management operation that will affect the operation of the server (e.g. changing a datasource) you can suspend the server, perform the operation, then resume the server. This allows all requests to finish, and makes sure that no requests are running while the management changes are taking place.

When a server is suspending it goes through four different phases:

- **RUNNING** - The normal state, the server is accepting requests and running normally
- **PRE_SUSPEND** - In PRE_SUSPEND the server will notify external parties that it is about to suspend, for example mod_cluster will notify the load balancer that the deployment is suspending. Requests are still accepted in this phase.
- **SUSPENDING** - All new requests are rejected, and the server is waiting for all active requests to finish. If there are no active requests at suspend time this phase will be skipped.
- **SUSPENDED** - All requests have completed, and the server is suspended.

Starting Suspended

In order to start into suspended mode when using a standalone server you need to add **--start-mode=suspend** to the command line. It is also possible to specify the start-mode in the **reload** operation to cause the server to reload into suspended mode (other possible values for start-mode are **normal** and **admin-only**).

In domain mode servers can be started in suspended mode by passing the **suspend=true** parameter to any command that causes a server to start, restart or reload (e.g. :start-servers(suspend=true)).

The Request Controller Subsystem

Wildfly introduces a new subsystem called the Request Controller Subsystem. This optional subsystem tracks all requests at their entry point, which how the graceful shutdown mechanism know when all requests are done (it also allows you to provide a global limit on the total number of running requests).

If this subsystem is not present suspend/resume will be limited, in general things that happen in the PRE_SUSPEND phase will work as normal (stopping message delivery, notifying the load balancer), however the server will not wait for all requests to complete and instead move straight to SUSPENDED mode.

There is a small performance penalty associated with the request controller subsystem (about on par with enabling statistics), so if you do not require the suspend/resume functionality this subsystem can be removed to get a small performance boost.

Subsystem Integrations

Suspend/Resume is a service provided by the Wildfly platform that any subsystem may choose to integrate with. Some subsystems integrate directly with the suspend controller, while others integrate through the request controller subsystem.

The following subsystems support graceful shutdown. Note that only subsystems that provide an external entry point to the server need graceful shutdown support, for example the JAX-RS subsystem does not require suspend/resume support as all access to JAX-RS is through the web connector.

- **Undertow** - Undertow will wait for all requests to finish
- **mod_cluster** - The mod_cluster subsystem will notify the load balancer that the server is suspending in the PRE_SUSPEND phase.
- **EJB** - EJB will wait for all remote EJB requests and MDB message deliveries to finish. Delivery to MDB's is stopped in the PRE_SUSPEND phase. EJB timers are suspended, and missed timers will be activated when the server is resumed.
- **Batch** - Batch jobs will be stopped at a checkpoint while the server is suspending. They will be restarted from that checkpoint when the server returns to running mode.
- **EE Concurrency** - The server will wait for all active jobs to finish. All jobs that have already been queued will be skipped.
- **Transactions** - transaction subsystem waits for all running transactions to finish while server is suspending. During that time server refuses to start any new transaction. But any in-flight transaction will be serviced - e.g. it means that server accepts any incoming remote call which carries context of the transaction already started at the suspending server.
  When you work with EJBs you have to enable the graceful shutdown functionality by setting attribute `enable-graceful-txn-shutdown` to `true`.
  (at the `ejb3 subsystem` xml, for example):
  `<enable-graceful-txn-shutdown value="false"/>`
  By **default** graceful shutdown it's **disabled** for ejb subsystem.
  The reason is that the behavior might be unwelcome in cluster environments, as the server notifies remote clients that the node is no longer available for remote calls only after the transactions are finished. During that brief window of time, the client of a cluster may send a new request to a node that is shutting down and will refuse the request because it is not related to an existing transaction.
  If this attribute `enable-graceful-txn-shutdown` is set to `false`, we disable the graceful behavior and EJB clients will not attempt to invoke the node when it suspends, regardless of active transactions.

Standalone Mode

Suspend/Resume can be controlled via the following CLI operations in standalone mode:

```
:suspend(timeout=z)
```

Suspends the server. If the timeout is specified it will wait up to the specified number of seconds for all requests to finish. If there is no timeout specified or the value is less than zero it will wait indefinitely.

```
:resume
```

Resumes a previously suspended server. The server should be able to begin serving requests immediately.

```
:read-attribute(name=suspend-state)
```

Returns the current suspend state of the server.

```
:shutdown(timeout=x)
```

If a timeout parameter is passed to the shutdown command then a graceful shutdown will be performed. The server will be suspended, and will wait up to the specified number of seconds for all requests to finish before shutting down. A timeout value of less than zero means it will wait indefinitely.

Domain Mode

Domain mode has similar commands as standalone mode, however they can be applied at both the global and server group levels:

Whole Domain

```
:suspend-servers(timeout=x)
```

:resume-servers

:stop-servers(timeout=x)

Server Group

```
/server-group=main-server-group:suspend-servers(timeout=x)
```

```
/server-group=main-server-group:resume-servers
```

```
/server-group=main-server-group:stop-servers(timeout=x)
```

Server

```
/host=master/server-config=server-one:suspend(timeout=x)
```

/host=master/server-config=server-one:resume

/host=master/server-config=server-one:stop(timeout=x)

5.19.3 Starting & stopping Servers in a Managed Domain

Starting a standalone server is done through the `bin/standalone.sh` script. However in a managed domain server instances are managed by the domain controller and need to be started through the management layer:

First of all, get to know which `servers` are configured on a particular `host`:

```
[domain@localhost:9990 /] :read-children-names(child-type=host)
{
   "outcome" => "success",
   "result" => ["local"]
}


[domain@localhost:9990 /] /host=local:read-children-names(child-type=server-config)
{
   "outcome" => "success",
   "result" => [
       "my-server",
       "server-one",
       "server-three"
   ]
}
```

Now that we know, that there are two `servers` configured on `host` "*local*", we can go ahead and check their status:

```
[domain@localhost:9990 /]
/host=local/server-config=server-one:read-resource(include-runtime=true)
{
   "outcome" => "success",
   "result" => {
       "auto-start" => true,
       "group" => "main-server-group",
       "interface" => undefined,
       "name" => "server-one",
       "path" => undefined,
       "socket-binding-group" => undefined,
       "socket-binding-port-offset" => undefined,
       "status" => "STARTED",
       "system-property" => undefined,
       "jvm" => {"default" => undefined}
   }
}
```

You can change the server state through the "*start*" and "*stop*" operations

```
[domain@localhost:9990 /] /host=local/server-config=server-one:stop
{
   "outcome" => "success",
   "result" => "STOPPING"
}
```

Navigating through the domain topology is much more simple when you use the web interface.

5.19.4 Controlling JVM settings

Configuration of the JVM settings is different for a managed domain and a standalone server. In a managed domain, the domain controller components are responsible for starting and stoping server processes and hence determine the JVM settings. For a standalone server, it's the responsibility of the process that started the server (e.g. passing them as command line arguments).

Managed Domain

In a managed domain the JVM settings can be declared at different scopes: For a specific server group, for a host or for a particular server. If not declared, the settings are inherited from the parent scope. This allows you to customize or extend the JVM settings within every layer.

Let's take a look at the JVM declaration for a server group:

```
<server-groups>
       <server-group name="main-server-group" profile="default">
           <jvm name="default">
               <heap size="64m" max-size="512m"/>
           </jvm>
           <socket-binding-group ref="standard-sockets"/>
       </server-group>
       <server-group name="other-server-group" profile="default">
           <jvm name="default">
               <heap size="64m" max-size="512m"/>
           </jvm>
           <socket-binding-group ref="standard-sockets"/>
       </server-group>
</server-groups>
```

(See `domain/configuration/domain.xml`)

In this example the server group "main-server-group" declares a heap size of `64m` and a maximum heap size of `512m`. Any server that belongs to this group will inherit these settings. You can change these settings for the group as a whole, or a specific server or host:

```
<servers>
       <server name="server-one" group="main-server-group" auto-start="true">
           <jvm name="default"/>
       </server>
       <server name="server-two" group="main-server-group" auto-start="true">
           <jvm name="default">
               <heap size="64m" max-size="256m"/>
           </jvm>
           <socket-binding-group ref="standard-sockets" port-offset="150"/>
       </server>
       <server name="server-three" group="other-server-group" auto-start="false">
           <socket-binding-group ref="standard-sockets" port-offset="250"/>
       </server>
</servers>
```

(See domain/configuration/host.xml)

In this case, *server-two*, belongs to the *main-server-group* and inherits the JVM settings named *default*, but declares a lower maximum heap size.

```
[domain@localhost:9999 /] /host=local/server-config=server-two/jvm=default:read-resource
{
   "outcome" => "success",
   "result" => {
       "heap-size" => "64m",
       "max-heap-size" => "256m",
   }
}
```

Standalone Server

For a standalone sever you have to pass in the JVM settings either as command line arguments when executing the `$JBOSS_HOME/bin/standalone.sh` script, or by declaring them in `$JBOSS_HOME/bin/standalone.conf`. (For Windows users, the script to execute is `%JBOSS_HOME%/bin/standalone.bat` while the JVM settings can be declared in `%JBOSS_HOME%/bin/standalone.conf.bat`.)

5.19.5 Administrative audit logging

WildFly comes with audit logging built in for management operations affecting the management model. By default it is turned off. The information is output as JSON records.

The default configuration of audit logging in standalone.xml looks as follows:

```
<management>
        <security-realms>
...
        </security-realms>
        <audit-log>
            <formatters>
                <json-formatter name="json-formatter"/>
            </formatters>
            <handlers>
                <file-handler name="file" formatter="json-formatter" path="audit-log.log"
relative-to="jboss.server.data.dir"/>
            </handlers>
            <logger log-boot="true" log-read-only="true" enabled="false">
                <handlers>
                    <handler name="file"/>
                </handlers>
            </logger>
        </audit-log>
...
```

Looking at this via the CLI it looks like

```
[standalone@localhost:9990 /]
/core-service=management/access=audit:read-resource(recursive=true)
{
    "outcome" => "success",
    "result" => {
        "file-handler" => {"file" => {
            "formatter" => "json-formatter",
            "max-failure-count" => 10,
            "path" => "audit-log.log",
            "relative-to" => "jboss.server.data.dir"
        }},
        "json-formatter" => {"json-formatter" => {
            "compact" => false,
            "date-format" => "yyyy-MM-dd HH:mm:ss",
            "date-separator" => " - ",
            "escape-control-characters" => false,
            "escape-new-line" => false,
            "include-date" => true
        }},
        "logger" => {"audit-log" => {
            "enabled" => false,
            "log-boot" => true,
            "log-read-only" => false,
            "handler" => {"file" => {}}
        }},
        "syslog-handler" => undefined
    }
}
```

To enable it via CLI you need just

```
[standalone@localhost:9990 /]
/core-service=management/access=audit/logger=audit-log:write-attribute(name=enabled,value=true)
{"outcome" => "success"}
```

Audit data are stored in standalone/data/audit-log.log.

The audit logging subsystem has a lot of internal dependencies, and it logs operations changing, enabling and disabling its components. When configuring or changing things at runtime it is a good idea to make these changes as part of a CLI batch. For example if you are adding a syslog handler you need to add the handler and its information as one step. Similarly if you are using a file handler, and want to change its `path` and `relative-to` attributes, that needs to happen as one step.

JSON Formatter

The first thing that needs configuring is the formatter, we currently support outputting log records as JSON. You can define several formatters, for use with different handlers. A log record has the following format, and it is the formatter's job to format the data presented:

```
2013-08-12 11:01:12 - {
    "type" : "core",
    "r/o" : false,
    "booting" : false,
    "version" : "8.0.0.Alpha4",
    "user" : "$local",
    "domainUUID" : null,
    "access" : "NATIVE",
    "remote-address" : "127.0.0.1/127.0.0.1",
    "success" : true,
    "ops" : [JMX|WFLY8:JMX subsystem configuration],
        "operation" : "write-attribute",
        "name" : "enabled",
        "value" : true,
        "operation-headers" : {"caller-type" : "user"}
    }]
}
```

It includes an optional timestamp and then the following information in the json record

Field name	Description
`type`	This can have the values `core`, meaning it is a management operation, or `jmx` meaning it comes from the jmx subsystem (see the jmx subsystem for configuration of the jmx subsystem's audit logging)
`r/o`	`true` if the operation does not change the management model, `false` otherwise
`booting`	`true` if the operation was executed during the bootup process, `false` if it was executed once the server is up and running
`version`	The version number of the WildFly instance
`user`	The username of the authenticated user. In this case the operation has been logged via the CLI on the same machine as the running server, so the special `$local` user is used
`domainUUID`	An ID to link together all operations as they are propagated from the Doman Controller to it servers, slave Host Controllers, and slave Host Controller servers
`access`	This can have one of the following values: *`NATIVE` - The operation came in through the native management interface, for example the CLI *`HTTP` - The operation came in through the domain HTTP interface, for example the admin console *`JMX` - The operation came in through the JMX subsystem. See JMX for how to configure audit logging for JMX.
`remote-address`	The address of the client executing this operation
`success`	`true` if the operation succeeded, `false` if it was rolled back
`ops`	The operations being executed. This is a list of the operations serialized to JSON. At boot this will be all the operations resulting from parsing the xml. Once booted the list will typically just contain a single entry

The json formatter resource has the following attributes:

Attribute	Description
`include-date`	Boolan toggling whether or not to include the timestamp in the formatted log records
`date-separator`	A string containing characters to separate the date and the rest of the formatted log message. Will be ignored if `include-date=false`
`date-format`	The date format to use for the timestamp as understood by `java.text.SimpleDateFormat`. Will be ignored if `include-date=false`
`compact`	If true will format the JSON on one line. There may still be values containing new lines, so if having the whole record on one line is important, set escape-new-line or escape-control-characters to true
`escape-control-characters`	If `true` it will escape all control characters (ascii entries with a decimal value < 32) with the ascii code in octal, e.g. a new line becomes '#012'. If this is `true`, it will override `escape-new-line=false`
`escape-new-line`	If `true` it will escape all new lines with the ascii code in octal, e.g. "#012".

Handlers

A handler is responsible for taking the formatted data and logging it to a location. There are currently two types of handlers, File and Syslog. You can configure several of each type of handler and use them to log information.

File handler

The file handlers log the audit log records to a file on the server. The attributes for the file handler are

Attribute	Description	Read Only
`formatter`	The name of a JSON formatter to use to format the log records	`false`
`path`	The path of the audit log file	`false`
`relative-to`	The name of another previously named path, or of one of the standard paths provided by the system. If `relative-to` is provided, the value of the `path` attribute is treated as relative to the path specified by this attribute	`false`
`failure-count`	The number of logging failures since the handler was initialized	`true`
`max-failure-count`	The maximum number of logging failures before disabling this handler	`false`
`disabled-due-to-failure`	`true` if this handler was disabled due to logging failures	`true`

In our standard configuration `path=audit-log.log` and `relative-to=jboss.server.data.dir`, typically this will be `$JBOSS_HOME/standalone/data/audit-log.log`

Syslog handler

The default configuration does not have syslog audit logging set up. Syslog is a better choice for audit logging since you can log to a remote syslog server, and secure the authentication to happen over TLS with client certificate authentication. Syslog servers vary a lot in their capabilities so not all settings in this section apply to all syslog servers. We have tested with rsyslog.

The address for the syslog handler is `/core-service=management/access=audit/syslog-handler=*` and just like file handlers you can add as many syslog entries as you like. The syslog handler resources reference the main RFC's for syslog a fair bit, for reference they can be found at:
*http://www.ietf.org/rfc/rfc3164.txt
*http://www.ietf.org/rfc/rfc5424.txt
*http://www.ietf.org/rfc/rfc6587.txt

The syslog handler resource has the following attributes:

`formatter`	The name of a JSON formatter to use to format the log records	`false`
`failure-count`	The number of logging failures since the handler was initialized	`true`
`max-failure-count`	The maximum number of logging failures before disabling this handler	`false`
`disabled-due-to-failure`	`true` if this handler was disabled due to logging failures	`true`
`syslog-format`	Whether to set the syslog format to the one specified in RFC-5424 or RFC-3164	`false`
`max-length`	The maximum length in bytes a log message, including the header, is allowed to be. If undefined, it will default to 1024 bytes if the syslog-format is RFC3164, or 2048 bytes if the syslog-format is RFC5424.	`false`
`truncate`	Whether or not a message, including the header, should truncate the message if the length in bytes is greater than the maximum length. If set to false messages will be split and sent with the same header values	`false`

When adding a syslog handler you also need to add the protocol it will use to communicate with the syslog server. The valid choices for protocol are `UDP`,`TCP` and `TLS`. The protocol must be added at the same time as you add the syslog handler, or it will fail. Also, you can only add one protocol for the handler.

UDP

Configures the handler to use UDP to communicate with the syslog server. The address of the `UDP` resource is `/core-service=management/access=audit/syslog-handler=*/protocol=udp`. The attributes of the `UDP` resource are:

Attribute	Description
`host`	The host of the syslog server for the udp requests
`port`	The port of the syslog server listening for the udp requests

TCP

Configures the handler to use TCP to communicate with the syslog server. The address of the `TCP` resource is `/core-service=management/access=audit/syslog-handler=*/protocol=tcp`. The attributes of the `TCP` resource are:

Attribute	Description
`host`	The host of the syslog server for the tcp requests
`port`	The port of the syslog server listening for the tcp requests
`message-transfer`	The message transfer setting as described in section 3.4 of RFC-6587. This can either be OCTET_COUNTING as described in section 3.4.1 of RFC-6587, or NON_TRANSPARENT_FRAMING as described in section 3.4.1 of RFC-6587

TLS

Configures the handler to use TLC to communicate securely with the syslog server. The address of the `TLS` resource is `/core-service=management/access=audit/syslog-handler=*/protocol=tls`. The attributes of the `TLS` resource are the same as for `TCP`:

Attribute	Description
`host`	The host of the syslog server for the tls requests
`port`	The port of the syslog server listening for the tls requests
`message-transfer`	The message transfer setting as described in section 3.4 of RFC-6587. This can either be OCTET_COUNTING as described in section 3.4.1 of RFC-6587, or NON_TRANSPARENT_FRAMING as described in section 3.4.1 of RFC-6587

If the syslog server's TLS certificate is not signed by a certificate signing authority, you will need to set up a truststore to trust the certificate. The resource for the trust store is a child of the `TLS` resource, and the full address is `/core-service=management/access=audit/syslog-handler=*/protocol=tls/authenticati`. The attributes of the truststore resource are:

Attribute	Description
`keystore-password`	The password for the truststore
`keystore-path`	The path of the truststore
`keystore-relative-to`	The name of another previously named path, or of one of the standard paths provided by the system. If `keystore-relative-to` is provided, the value of the `keystore-path` attribute is treated as relative to the path specified by this attribute

TLS with Client certificate authentication.
If you have set up the syslog server to require client certificate authentication, when creating your handler you will also need to set up a client certificate store containing the certificate to be presented to the syslog server. The address of the client certificate store resource is `/core-service=management/access=audit/syslog-handler=*/protocol=tls/authenticati` and its attributes are:

Attribute	Description
`keystore-password`	The password for the keystore
`key-password`	The password for the keystore key
`keystore-path`	The path of the keystore
`keystore-relative-to`	The name of another previously named path, or of one of the standard paths provided by the system. If `keystore-relative-to` is provided, the value of the `keystore-path` attribute is treated as relative to the path specified by this attribute

Logger configuration

The final part that needs configuring is the logger for the management operations. This references one or more handlers and is configured at `/core-service=management/access=audit/logger=audit-log`. The attributes for this resource are:

Attribute	Description
`enabled`	`true` to enable logging of the management operations
`log-boot`	`true` to log the management operations when booting the server, `false` otherwise
`log-read-only`	If `true` all operations will be audit logged, if `false` only operations that change the model will be logged

Then which handlers are used to log the management operations are configured as `handler=*` children of the logger.

Domain Mode (host specific configuration)

In domain mode audit logging is configured for each host in its `host.xml` file. This means that when connecting to the DC, the configuration of the audit logging is under the host's entry, e.g. here is the default configuration:

```
[domain@localhost:9990 /]
/host=master/core-service=management/access=audit:read-resource(recursive=true)
{
    "outcome" => "success",
    "result" => {
        "file-handler" => {
            "host-file" => {
                "formatter" => "json-formatter",
                "max-failure-count" => 10,
                "path" => "audit-log.log",
                "relative-to" => "jboss.domain.data.dir"
            },
            "server-file" => {
                "formatter" => "json-formatter",
                "max-failure-count" => 10,
                "path" => "audit-log.log",
                "relative-to" => "jboss.server.data.dir"
            }
        },
        "json-formatter" => {"json-formatter" => {
            "compact" => false,
            "date-format" => "yyyy-MM-dd HH:mm:ss",
            "date-separator" => " - ",
            "escape-control-characters" => false,
            "escape-new-line" => false,
            "include-date" => true
        }},
        "logger" => {"audit-log" => {
            "enabled" => false,
            "log-boot" => true,
            "log-read-only" => false,
            "handler" => {"host-file" => {}}
        }},
        "server-logger" => {"audit-log" => {
            "enabled" => false,
            "log-boot" => true,
            "log-read-only" => false,
            "handler" => {"server-file" => {}}
        }},
        "syslog-handler" => undefined
    }
}
```

We now have two file handlers, one called `host-file` used to configure the file to log management operations on the host, and one called `server-file` used to log management operations executed on the servers. Then `logger=audit-log` is used to configure the logger for the host controller, referencing the `host-file` handler. `server-logger=audit-log` is used to configure the logger for the managed servers, referencing the `server-file` handler. The attributes for `server-logger=audit-log` are the same as for `server-logger=audit-log` in the previous section. Having the host controller and server loggers configured independently means we can control audit logging for managed servers and the host controller independently.

5.19.6 Canceling management operations

WildFly includes the ability to use the CLI to cancel management requests that are not proceeding normally.

The cancel-non-progressing-operation operation

The `cancel-non-progressing-operation` operation instructs the target process to find any operation that isn't proceeding normally and cancel it.

On a standalone server:

```
[standalone@localhost:9990 /]
/core-service=management/service=management-operations:cancel-non-progressing-operation
{
    "outcome" => "success",
    "result" => "-1155777943"
}
```

The result value is an internal identification number for the operation that was cancelled.

On a managed domain host controller, the equivalent resource is in the host=<hostname> portion of the management resource tree:

```
[domain@localhost:9990 /]
/host=host-a/core-service=management/service=management-operations:cancel-non-progressing-operatio
"outcome" => "success",
    "result" => "2156877946"
}
```

An operation can be cancelled on an individual managed domain server as well:

```
[domain@localhost:9990 /]
/host=host-a/server=server-one/core-service=management/service=management-operations:cancel-non-pr
"outcome" => "success",
    "result" => "6497786512"
}
```

An operation is considered to be not proceeding normally if it has been executing with the exclusive operation lock held for longer than 15 seconds. Read-only operations do not acquire the exclusive operation lock, so this operation will not cancel read-only operations. Operations blocking waiting for another operation to release the exclusive lock will also not be cancelled.

If there isn't any operation that is failing to proceed normally, there will be a failure response:

```
[standalone@localhost:9990 /]
/core-service=management/service=management-operations:cancel-non-progressing-operation
{
    "outcome" => "failed",
    "failure-description" => "WFLYDM0089: No operation was found that has been holding the
operation execution write lock for long than [15] seconds",
    "rolled-back" => true
}
```

The find-non-progressing-operation operation

To simply learn the id of an operation that isn't proceeding normally, but not cancel it, use the `find-non-progressing-operation` operation:

```
[standalone@localhost:9990 /]
/core-service=management/service=management-operations:find-non-progressing-operation
{
    "outcome" => "success",
    "result" => "-1155777943"
}
```

If there is no non-progressing operation, the outcome will still be `success` but the result will be `undefined`.

Once the id of the operation is known, the management resource for the operation can be examined to learn more about its status.

Examining the status of an active operation

There is a management resource for any currently executing operation that can be queried:

```
[standalone@localhost:9990 /]
/core-service=management/service=management-operations/active-operation=-1155777943:read-resource(
"outcome" => "success",
    "result" => {
        "access-mechanism" => "undefined",
        "address" => [
            ("deployment" => "example")
        ],
        "caller-thread" => "management-handler-thread - 24",
        "cancelled" => false,
        "exclusive-running-time" => 101918273645L,
        "execution-status" => "awaiting-stability",
        "operation" => "deploy",
        "running-time" => 101918279999L
    }
}
```

The response includes the following attributes:

Field	Meaning
access-mechanism	The mechanism used to submit a request to the server. NATIVE, JMX, HTTP
address	The address of the resource targeted by the operation. The value in the final element of the address will be '<hidden>' if the caller is not authorized to address the operation's target resource.
caller-thread	The name of the thread that is executing the operation.
cancelled	Whether the operation has been cancelled.
exclusive-running-time	Amount of time in nanoseconds the operation has been executing with the exclusive operation execution lock held, or -1 if the operation does not hold the exclusive execution lock.
execution-status	The current activity of the operation. See below for details.
operation	The name of the operation, or '<hidden>' if the caller is not authorized to address the operation's target resource.
running-time	Amount of time the operation has been executing, in nanoseconds.

The following are the values for the `execution-status` attribute:

Value	Meaning
executing	The caller thread is actively executing
awaiting-other-operation	The caller thread is blocking waiting for another operation to release the exclusive execution lock
awaiting-stability	The caller thread has made changes to the service container and is waiting for the service container to stabilize
completing	The operation is committed and is completing execution
rolling-back	The operation is rolling back

All currently executing operations can be viewed in one request using the `read-children-resources` operation:

```
[standalone@localhost:9990 /]
/core-service=management/service=management-operations:read-children-resources(child-type=active-o
"outcome" => "success",
    "result" => {"-1155777943" => {
        "access-mechanism" => "undefined",
        "address" => [
            ("deployment" => "example")
        ],
        "caller-thread" => "management-handler-thread - 24",
        "cancelled" => false,
        "exclusive-running-time" => 101918273645L,
        "execution-status" => "awaiting-stability",
        "operation" => "deploy",
        "running-time" => 101918279999L
    },
    {"-1246693202" => {
        "access-mechanism" => "undefined",
        "address" => [
            ("core-service" => "management"),
            ("service" => "management-operations")
        ],
        "caller-thread" => "management-handler-thread - 30",
        "cancelled" => false,
        "exclusive-running-time" => -1L,
        "execution-status" => "executing",
        "operation" => "read-children-resources",
        "running-time" => 3356000L
    }}
}
```

Canceling a specific operation

The `cancel-non-progressing-operation` operation is a convenience operation for identifying and canceling an operation. However, an administrator can examine the active-operation resources to identify any operation, and then directly cancel it by invoking the `cancel` operation on the resource for the desired operation.

```
[standalone@localhost:9990 /]
/core-service=management/service=management-operations/active-operation=-1155777943:cancel
{
    "outcome" => "success",
    "result" => undefined
}
```

Controlling operation blocking time

As an operation executes, the execution thread may block at various points, particularly while waiting for the service container to stabilize following any changes. Since an operation may be holding the exclusive execution lock while blocking, in WildFly execution behavior was changed to ensure that blocking will eventually time out, resulting in roll back of the operation.

The default blocking timeout is 300 seconds. This is intentionally long, as the idea is to only trigger a timeout when something has definitely gone wrong with the operation, without any false positives.

An administrator can control the blocking timeout for an individual operation by using the `blocking-timeout` operation header. For example, if a particular deployment is known to take an extremely long time to deploy, the default 300 second timeout could be increased:

```
[standalone@localhost:9990 /] deploy /tmp/mega.war --headers={blocking-timeout=450}
```

Note the blocking timeout is **not** a guaranteed maximum execution time for an operation. If it only a timeout that will be enforced at various points during operation execution.

5.19.7 Configuration file history

The management operations may modify the model. When this occurs the xml backing the model is written out again reflecting the latest changes. In addition a full history of the file is maintained. The history of the file goes in a separate directory under the configuration directory.

As mentioned in Command line parameters#parameters the default configuration file can be selected using a command-line parameter. For a standalone server instance the history of the active `standalone.xml` is kept in `jboss.server.config.dir`/standalone_xml_history (See Command line parameters#standalone_system_properties for more details). For a domain the active `domain.xml` and `host.xml` histories are kept in `jboss.domain.config.dir`/domain_xml_history and `jboss.domain.config.dir`/host_xml_history.

The rest of this section will only discuss the history for `standalone.xml`. The concepts are exactly the same for `domain.xml` and `host.xml`.

Within `standalone_xml_history` itself following a successful first time boot we end up with three new files:

- `standalone.initial.xml` - This contains the original configuration that was used the first time we successfully booted. This file will never be overwritten. You may of course delete the history directory and any files in it at any stage.
- `standalone.boot.xml` - This contains the original configuration that was used for the last successful boot of the server. This gets overwritten every time we boot the server successfully.
- `standalone.last.xml` - At this stage the contents will be identical to `standalone.boot.xml`. This file gets overwritten each time the server successfully writes the configuration, if there was an unexpected failure writing the configuration this file is the last known successful write.

`standalone_xml_history` contains a directory called `current` which should be empty. Now if we execute a management operation that modifies the model, for example adding a new system property using the CLI:

```
[standalone@localhost:9990 /] /system-property=test:add(value="test123")
{"outcome" => "success"}
```

What happens is:

- The original configuration file is backed up to `standalone_xml_history/current/standalone.v1.xml`. The next change to the model would result in a file called `standalone.v2.xml` etc. The 100 most recent of these files are kept.
- The change is applied to the original configuration file
- The changed original configuration file is copied to `standalone.last.xml`

When restarting the server, any existing `standalone_xml_history/current` directory is moved to a new timestamped folder within the `standalone_xml_history`, and a new `current` folder is created. These timestamped folders are kept for 30 days.

Snapshots

In addition to the backups taken by the server as described above you can manually take take snapshots which will be stored in the `snapshot` folder under the `_xml_history` folder, the automatic backups described above are subject to automatic house keeping so will eventually be automatically removed, the snapshots on the other hand can be entirely managed by the administrator.

You may also take your own snapshots using the CLI:

```
[standalone@localhost:9990 /] :take-snapshot
{
    "outcome" => "success",
    "result" => {"name" =>
"/Users/kabir/wildfly/standalone/configuration/standalone_xml_history/snapshot/20110630-172258657s
```

You can also use the CLI to list all the snapshots

```
[standalone@localhost:9990 /] :list-snapshots
{
    "outcome" => "success",
    "result" => {
        "directory" =>
"/Users/kabir/wildfly/standalone/configuration/standalone_xml_history/snapshot",
        "names" => [
            "20110630-165714239standalone.xml",
            "20110630-165821795standalone.xml",
            "20110630-170113581standalone.xml",
            "20110630-171411463standalone.xml",
            "20110630-171908397standalone.xml",
            "20110630-172258657standalone.xml"
        ]
    }
}
```

To delete a particular snapshot:

```
[standalone@localhost:9990 /] :delete-snapshot(name="20110630-165714239standalone.xml")
{"outcome" => "success"}
```

and to delete all snapshots:

```
[standalone@localhost:9990 /] :delete-snapshot(name="all")
{"outcome" => "success"}
```

In domain mode executing the snapshot operations against the root node will work against the domain model. To do this for a host model you need to navigate to the host in question:

```
[domain@localhost:9990 /] /host=master:list-snapshots
{
    "outcome" => "success",
    "result" => {
        "domain-results" => {"step-1" => {
            "directory" =>
"/Users/kabir/wildfly/domain/configuration/host_xml_history/snapshot",
            "names" => [
                "20110630-141129571host.xml",
                "20110630-172522225host.xml"
            ]
        }},
        "server-operations" => undefined
    }
}
```

Subsequent Starts

For subsequent server starts it may be desirable to take the state of the server back to one of the previously known states, for a number of items an abbreviated reverence to the file can be used:

Abreviation	Parameter	Description
`initial`	`--server-config=initial`	This will start the server using the initial configuration first used to start the server.
`boot`	`--server-config=boot`	This will use the configuration from the last successful boot of the server.
`last`	`--server-config=last`	This will start the server using the configuration backed up from the last successful save.
`v?`	`--server-config=v?`	This will server the _xml_history/current folder for the configuration where ? is the number of the backup to use.
`-?`	`--server-config=-?`	The server will be started after searching the snapshot folder for the configuration which matches this prefix.

In addition to this the `--server-config` parameter can always be used to specify a configuration relative to the `jboss.server.config.dir` and finally if no matching configuration is found an attempt to locate the configuration as an absolute path will be made.

5.19.8 Application deployment

Managed Domain

In a managed domain, deployments are associated with a `server-group` (see Core management concepts). Any server within the server group will then be provided with that deployment.

The domain and host controller components manage the distribution of binaries across network boundaries.

Deployment Commands

Distributing deployment binaries involves two steps: uploading the deployment to the repository the domain controller will use to distribute its contents, and then assigning the deployment to one or more server groups.

You can do this in one sweep with the CLI:

```
[domain@localhost:9990 /] deploy ~/Desktop/test-application.war
Either --all-server-groups or --server-groups must be specified.

[domain@localhost:9990 /] deploy ~/Desktop/test-application.war --all-server-groups
'test-application.war' deployed successfully.
```

The deployment will be available to the domain controller, assigned to a server group, and deployed on all running servers in that group:

```
[domain@localhost:9990 /] :read-children-names(child-type=deployment)
{
   "outcome" => "success",
   "result" => [
       "mysql-connector-java-5.1.15.jar",
       "test-application.war"
   ]
}

[domain@localhost:9990 /]
/server-group=main-server-group/deployment=test-application.war:read-resource(include-runtime)
{
   "outcome" => "success",
   "result" => {
       "enabled" => true,
       "name" => "test-application.war",
       "managed" => true,
       "runtime-name" => "test-application.war"
   }
}
```

If you only want the deployment deployed on servers in some server groups, but not all, use the `--server-groups` parameter instead of `-all-server-groups`:

```
[domain@localhost:9990 /] deploy ~/Desktop/test-application.war
--server-groups=main-server-group,another-group
'test-application.war' deployed successfully.
```

If you have a new version of the deployment that you want to deploy replacing an existing one, use the `--force` parameter:

```
[domain@localhost:9990 /] deploy ~/Desktop/test-application.war --all-server-groups --force
'test-application.war' deployed successfully.
```

You can remove binaries from server groups with the `undeploy` command:

```
[domain@localhost:9990 /] undeploy test-application.war --all-relevant-server-groups
Successfully undeployed test-application.war.

[domain@localhost:9990 /]
/server-group=main-server-group:read-children-names(child-type=deployment)
{
   "outcome" => "success",
   "result" => []
}
```

If you only want to undeploy from some server groups but not others, use the `-server-groups` parameter instead of `-all-relevant-server-groups`.

The CLI `deploy` command supports a number of other parameters that can control behavior. Use the `--help` parameter to learn more:

```
[domain@localhost:9990 /] deploy --help
[...]
```


Managing deployments through the web interface provides an alternate, sometimes simpler approach.

Exploded managed deployments

Managed and unmanaged deployments can be 'exploded', i.e. on the filesystem in the form of a directory structure whose structure corresponds to an unzipped version of the archive. An exploded deployment can be convenient to administer if your administrative processes involve inserting or replacing files from a base version in order to create a version tailored for a particular use (for example, copy in a base deployment and then copy in a jboss-web.xml file to tailor a deployment for use in WildFly.) Exploded deployments are also nice in some development scenarios, as you can replace static content (e.g. .html, .css) files in the deployment and have the new content visible immediately without requiring a redeploy.

Since unmanaged deployment content is directly in your charge, the following operations only make sense for a managed deployment.

```
[domain@localhost:9990 /] /deployment=exploded.war:add(content=[{empty=true}])
```

This will create an empty exploded deployment to which you'll be able to add content. The **empty** content parameter is required to check that you really intend to create an empty deployment and not just forget to define the content.

```
[domain@localhost:9990 /] /deployment=kitchensink.ear:explode()
```

This will 'explode' an existing archive deployment to its exploded format. This operation is not recursive so you need to explode the sub-deployment if you want to be able to manipulate the sub-deployment content. You can do this by specifying the sub-deployment archive **path** as a parameter to the explode operation.

```
[domain@localhost:9990 /]
/deployment=kitchensink.ear:explode(path=wildfly-kitchensink-ear-web.war)
```

Now you can add or remove content to your exploded deployment. Note that per-default this will overwrite existing contents, you can specify the overwrite parameter to make the operation fail if the content already exists.

```
[domain@localhost:9990 /]
/deployment=exploded.war:add-content(content=[{target-path=WEB-INF/classes/org/jboss/as/test/deplo
input-stream-index=/home/demo/org/jboss/as/test/deployment/trivial/ServiceActivatorDeployment.clas
{target-path=META-INF/MANIFEST.MF, input-stream-index=/home/demo/META-INF/MANIFEST.MF},
{target-path=META-INF/services/org.jboss.msc.service.ServiceActivator,
input-stream-index=/home/demo/META-INF/services/org.jboss.msc.service.ServiceActivator}])
```

Each content specifies a source content and the target path to which it will be copied relative to the deployment root. With WildFly 11 you can use **input-stream-index** (which was a convenient way to pass a stream of content) from the CLI by pointing it to a local file.

```
[domain@localhost:9990 /]
/deployment=exploded.war:remove-content(paths=[WEB-INF/classes/org/jboss/as/test/deployment/trivia
META-INF/MANIFEST.MF, META-INF/services/org.jboss.msc.service.ServiceActivator])
```

Now you can list the content of an exploded deployment, or just some part of it.

```
[domain@localhost:9990 /] /deployment=kitchensink.ear:browse-content(archive=false,
path=wildfly-kitchensink-ear-web.war)
{
    "outcome" => "success",
    "result" => [
        {
            "path" => "META-INF/",
            "directory" => true
        },
        {
            "path" => "META-INF/MANIFEST.MF",
            "directory" => false,
            "file-size" => 128L
        },
        {
            "path" => "WEB-INF/",
            "directory" => true
        },
        {
            "path" => "WEB-INF/templates/",
            "directory" => true
        },
        {
            "path" => "WEB-INF/classes/",
            "directory" => true
        },
        {
            "path" => "WEB-INF/classes/org/",
            "directory" => true
        },
        {
            "path" => "WEB-INF/classes/org/jboss/",
            "directory" => true
        },
        {
```

```
            "path" => "WEB-INF/classes/org/jboss/as/",
            "directory" => true
        },
        {
            "path" => "WEB-INF/classes/org/jboss/as/quickstarts/",
            "directory" => true
        },
        {
            "path" => "WEB-INF/classes/org/jboss/as/quickstarts/kitchensink_ear/",
            "directory" => true
        },
        {
            "path" => "WEB-INF/classes/org/jboss/as/quickstarts/kitchensink_ear/controller/",
            "directory" => true
        },
        {
            "path" => "WEB-INF/classes/org/jboss/as/quickstarts/kitchensink_ear/rest/",
            "directory" => true
        },
        {
            "path" => "WEB-INF/classes/org/jboss/as/quickstarts/kitchensink_ear/util/",
            "directory" => true
        },
        {
            "path" => "resources/",
            "directory" => true
        },
        {
            "path" => "resources/css/",
            "directory" => true
        },
        {
            "path" => "resources/gfx/",
            "directory" => true
        },
        {
            "path" => "WEB-INF/templates/default.xhtml",
            "directory" => false,
            "file-size" => 2113L
        },
        {
            "path" => "WEB-INF/faces-config.xml",
            "directory" => false,
            "file-size" => 1365L
        },
        {
            "path" =>
"WEB-INF/classes/org/jboss/as/quickstarts/kitchensink_ear/controller/MemberController.class",
            "directory" => false,
            "file-size" => 2750L
        },
        {
            "path" =>
"WEB-INF/classes/org/jboss/as/quickstarts/kitchensink_ear/rest/MemberResourceRESTService.class",
            "directory" => false,
            "file-size" => 6363L
        },
        {
```

```
            "path" =>
"WEB-INF/classes/org/jboss/as/quickstarts/kitchensink_ear/rest/JaxRsActivator.class",
            "directory" => false,
            "file-size" => 464L
        },
        {
            "path" =>
"WEB-INF/classes/org/jboss/as/quickstarts/kitchensink_ear/util/WebResources.class",
            "directory" => false,
            "file-size" => 667L
        },
        {
            "path" => "WEB-INF/beans.xml",
            "directory" => false,
            "file-size" => 1262L
        },
        {
            "path" => "index.xhtml",
            "directory" => false,
            "file-size" => 3603L
        },
        {
            "path" => "index.html",
            "directory" => false,
            "file-size" => 949L
        },
        {
            "path" => "resources/css/screen.css",
            "directory" => false,
            "file-size" => 4025L
        },
        {
            "path" => "resources/gfx/headerbkg.png",
            "directory" => false,
            "file-size" => 1147L
        },
        {
            "path" => "resources/gfx/asidebkg.png",
            "directory" => false,
            "file-size" => 1374L
        },
        {
            "path" => "resources/gfx/banner.png",
            "directory" => false,
            "file-size" => 41473L
        },
        {
            "path" => "resources/gfx/bkg-blkheader.png",
            "directory" => false,
            "file-size" => 116L
        },
        {
            "path" => "resources/gfx/rhjb_eap_logo.png",
            "directory" => false,
            "file-size" => 2637L
        },
        {
            "path" => "META-INF/maven/",
```

```
            "directory" => true
        },
        {
            "path" => "META-INF/maven/org.wildfly.quickstarts/",
            "directory" => true
        },
        {
            "path" => "META-INF/maven/org.wildfly.quickstarts/wildfly-kitchensink-ear-web/",
            "directory" => true
        },
        {
            "path" =>
"META-INF/maven/org.wildfly.quickstarts/wildfly-kitchensink-ear-web/pom.xml",
            "directory" => false,
            "file-size" => 4128L
        },
        {
            "path" =>
"META-INF/maven/org.wildfly.quickstarts/wildfly-kitchensink-ear-web/pom.properties",
            "directory" => false,
            "file-size" => 146L
        }
    ]
}
```

You also have a **read-content** operation but since it returns a binary stream, this is not displayable from the CLI.

```
[domain@localhost:9990 /] /deployment=kitchensink.ear:read-content(path=META-INF/MANIFEST.MF)
{
    "outcome" => "success",
    "result" => {"uuid" => "b373d587-72ee-4b1e-a02a-71fbb0c85d32"},
    "response-headers" => {"attached-streams" => [{
        "uuid" => "b373d587-72ee-4b1e-a02a-71fbb0c85d32",
        "mime-type" => "text/plain"
    }]}
}
```

The management CLI however provides high level commands to display or save binary stream attachments:

```
[domain@localhost:9990 /] attachment display
--operation=/deployment=kitchensink.ear:read-content(path=META-INF/MANIFEST.MF)
ATTACHMENT d052340a-abb7-4a66-aa24-4eeeb6b256be:
Manifest-Version: 1.0
Archiver-Version: Plexus Archiver
Built-By: mjurc
Created-By: Apache Maven 3.3.9
Build-Jdk: 1.8.0_91
```

```
[domain@localhost:9990 /] attachment save
--operation=/deployment=kitchensink.ear:read-content(path=META-INF/MANIFEST.MF) --file=example
File saved to /home/mjurc/wildfly/build/target/wildfly-11.0.0.Alpha1-SNAPSHOT/example
```

XML Configuration File

When you deploy content, the domain controller adds two types of entries to the `domain.xml` configuration file, one showing global information about the deployment, and another for each relevant server group showing how it is used by that server group:

```
[...]
<deployments>
   <deployment name="test-application.war"
               runtime-name="test-application.war">
       <content sha1="dda9881fa7811b22f1424b4c5acccb13c71202bd"/>
   </deployment>
</deployments>
[...]
<server-groups>
   <server-group name="main-server-group" profile="default">
       [...]
       <deployments>
           <deployment name="test-application.war" runtime-name="test-application.war"/>
       </deployments>
   </server-group>
</server-groups>
[...]
```

(See domain/configuration/domain.xml)

Standalone Server

Deployments on a standalone server work in a similar way to those on managed domains. The main difference is that there are no server group associations.

Deployment Commands

The same CLI commands used for managed domains work for standalone servers when deploying and removing an application:

```
[standalone@localhost:9990 /] deploy ~/Desktop/test-application.war
'test-application.war' deployed successfully.

[standalone@localhost:9990 /] undeploy test-application.war
Successfully undeployed test-application.war.
```

Deploying Using the Deployment Scanner

Deployment content (for example, war, ear, jar, and sar files) can be placed in the standalone/deployments directory of the WildFly distribution, in order to be automatically deployed into the server runtime. For this to work the `deployment-scanner` subsystem must be present. The scanner periodically checks the contents of the deployments directory and reacts to changes by updating the server.

Users are encouraged to use the WildFly management APIs to upload and deploy deployment content instead of relying on the deployment scanner that periodically scans the directory, particularly if running production systems.

Deployment Scanner Modes

The WildFly filesystem deployment scanner operates in one of two different modes, depending on whether it will directly monitor the deployment content in order to decide to deploy or redeploy it.

Auto-deploy mode:

The scanner will directly monitor the deployment content, automatically deploying new content and redeploying content whose timestamp has changed. This is similiar to the behavior of previous AS releases, although there are differences:

- A change in any file in an exploded deployment triggers redeploy. Because EE 6+ applications do not require deployment descriptors,
  there is no attempt to monitor deployment descriptors and only redeploy when a deployment descriptor changes.
- The scanner will place marker files in this directory as an indication of the status of its attempts to deploy or undeploy content. These are detailed below.

Manual deploy mode:

The scanner will not attempt to directly monitor the deployment content and decide if or when the end user wishes the content to be deployed. Instead, the scanner relies on a system of marker files, with the user's addition or removal of a marker file serving as a sort of command telling the scanner to deploy, undeploy or redeploy content.

Auto-deploy mode and manual deploy mode can be independently configured for zipped deployment content and exploded deployment content. This is done via the "auto-deploy" attribute on the deployment-scanner element in the standalone.xml configuration file:

```
<deployment-scanner scan-interval="5000" relative-to="jboss.server.base.dir"
   path="deployments" auto-deploy-zipped="true" auto-deploy-exploded="false"/>
```

By default, auto-deploy of zipped content is enabled, and auto-deploy of exploded content is disabled. Manual deploy mode is strongly recommended for exploded content, as exploded content is inherently vulnerable to the scanner trying to auto-deploy partially copied content.

Marker Files

The marker files always have the same name as the deployment content to which they relate, but with an additional file suffix appended. For example, the marker file to indicate the example.war file should be deployed is named example.war.dodeploy. Different marker file suffixes have different meanings.

The relevant marker file types are:

File	Purpose
.dodeploy	Placed by the user to indicate that the given content should be deployed into the runtime (or redeployed if already deployed in the runtime.)
.skipdeploy	Disables auto-deploy of the content for as long as the file is present. Most useful for allowing updates to exploded content without having the scanner initiate redeploy in the middle of the update. Can be used with zipped content as well, although the scanner will detect in-progress changes to zipped content and wait until changes are complete.
.isdeploying	Placed by the deployment scanner service to indicate that it has noticed a .dodeploy file or new or updated auto-deploy mode content and is in the process of deploying the content. This marker file will be deleted when the deployment process completes.
.deployed	Placed by the deployment scanner service to indicate that the given content has been deployed into the runtime. If an end user deletes this file, the content will be undeployed.
.failed	Placed by the deployment scanner service to indicate that the given content failed to deploy into the runtime. The content of the file will include some information about the cause of the failure. Note that with auto-deploy mode, removing this file will make the deployment eligible for deployment again.
.isundeploying	Placed by the deployment scanner service to indicate that it has noticed a .deployed file has been deleted and the content is being undeployed. This marker file will be deleted when the undeployment process completes.
.undeployed	Placed by the deployment scanner service to indicate that the given content has been undeployed from the runtime. If an end user deletes this file, it has no impact.
.pending	Placed by the deployment scanner service to indicate that it has noticed the need to deploy content but has not yet instructed the server to deploy it. This file is created if the scanner detects that some auto-deploy content is still in the process of being copied or if there is some problem that prevents auto-deployment. The scanner will not instruct the server to deploy or undeploy any content (not just the directly affected content) as long as this condition holds.

Basic workflows:

All examples assume variable $JBOSS_HOME points to the root of the WildFly distribution.

A) Add new zipped content and deploy it:

1. cp target/example.war/ $JBOSS_HOME/standalone/deployments
2. (Manual mode only) touch $JBOSS_HOME/standalone/deployments/example.war.dodeploy

B) Add new unzipped content and deploy it:

1. cp -r target/example.war/ $JBOSS_HOME/standalone/deployments
2. (Manual mode only) touch $JBOSS_HOME/standalone/deployments/example.war.dodeploy

C) Undeploy currently deployed content:

1. rm $JBOSS_HOME/standalone/deployments/example.war.deployed

D) Auto-deploy mode only: Undeploy currently deployed content:

1. rm $JBOSS_HOME/standalone/deployments/example.war

E) Replace currently deployed zipped content with a new version and deploy it:

1. cp target/example.war/ $JBOSS_HOME/standalone/deployments
2. (Manual mode only) touch $JBOSS_HOME/standalone/deployments/example.war.dodeploy

F) Manual mode only: Replace currently deployed unzipped content with a new version and deploy it:

1. rm $JBOSS_HOME/standalone/deployments/example.war.deployed
2. wait for $JBOSS_HOME/standalone/deployments/example.war.undeployed file to appear
3. cp -r target/example.war/ $JBOSS_HOME/standalone/deployments
4. touch $JBOSS_HOME/standalone/deployments/example.war.dodeploy

G) Auto-deploy mode only: Replace currently deployed unzipped content with a new version and deploy it:

1. touch $JBOSS_HOME/standalone/deployments/example.war.skipdeploy
2. cp -r target/example.war/ $JBOSS_HOME/standalone/deployments
3. rm $JBOSS_HOME/standalone/deployments/example.war.skipdeploy

H) Manual mode only: Live replace portions of currently deployed unzipped content without redeploying:

1. cp -r target/example.war/foo.html $JBOSS_HOME/standalone/deployments/example.war

I) Auto-deploy mode only: Live replace portions of currently deployed unzipped content without redeploying:

1. touch $JBOSS_HOME/standalone/deployments/example.war.skipdeploy
2. cp -r target/example.war/foo.html $JBOSS_HOME/standalone/deployments/example.war

J) Manual or auto-deploy mode: Redeploy currently deployed content (i.e. bounce it with no content change):

1. touch $JBOSS_HOME/standalone/deployments/example.war.dodeploy

K) Auto-deploy mode only: Redeploy currently deployed content (i.e. bounce it with no content change):

1. touch $JBOSS_HOME/standalone/deployments/example.war

The above examples use Unix shell commands. Windows equivalents are:

cp src dest --> xcopy /y src dest
cp -r src dest --> xcopy /e /s /y src dest
rm afile --> del afile
touch afile --> echo>> afile

Note that the behavior of 'touch' and 'echo' are different but the differences are not relevant to the usages in the examples above.

Managed and Unmanaged Deployments

WildFly supports two mechanisms for dealing with deployment content – managed and unmanaged deployments.

With a managed deployment the server takes the deployment content and copies it into an internal content repository and thereafter uses that copy of the content, not the original user-provided content. The server is thereafter responsible for the content it uses.

With an unmanaged deployment the user provides the local filesystem path of deployment content, and the server directly uses that content. However the user is responsible for ensuring that content, e.g. for making sure that no changes are made to it that will negatively impact the functioning of the deployed application.

To help you differentiate managed from unmanaged deployments the deployment model has a runtime boolean attribute 'managed'.

Managed deployments have a number of benefits over unmanaged:

- They can be manipulated by remote management clients, not requiring access to the server filesystem.
- In a managed domain, WildFly/EAP will take responsibility for replicating a copy of the deployment to all hosts/servers in the domain where it is needed. With an unmanaged deployment, it is the user's responsibility to have the deployment available on the local filesystem on all relevant hosts, at a consistent path.
- The deployment content actually used is stored on the filesystem in the internal content repository, which should help shelter it from unintended changes.

All of the previous examples above illustrate using managed deployments, except for any discussion of deployment scanner handling of exploded deployments. In WildFly 10 and earlier exploded deployments are always unmanaged, this is no longer the case since WildFly 11.

Content Repository

For a managed deployment, the actual file the server uses when creating runtime services is not the file provided to the CLI `deploy` command or to the web console. It is a copy of that file stored in an internal content repository. The repository is located in the `domain/data/content` directory for a managed domain, or in `standalone/data/content` for a standalone server. Actual binaries are stored in a subdirectory:

```
ls domain/data/content/
  |---/47
  |-----95cc29338b5049e238941231b36b3946952991
  |---/dd
  |-----a9881fa7811b22f1424b4c5acccb13c71202bd
```

The location of the content repository and its internal structure is subject to change at any time and should not be relied upon by end users.

The description of a managed deployment in the domain or standalone configuration file includes an attribute recording the SHA1 hash of the deployment content:

```
<deployments>
   <deployment name="test-application.war"
               runtime-name="test-application.war">
       <content sha1="dda9881fa7811b22f1424b4c5acccb13c71202bd"/>
   </deployment>
</deployments>
```

The WildFly process calculates and records that hash when the user invokes a management operation (e.g. CLI `deploy` command or using the console) providing deployment content. The user is not expected to calculate the hash.

The sha1 attribute in the content element tells the WildFly process where to find the deployment content in its internal content repository.

In a domain each host will have a copy of the content needed by its servers in its own local content repository. The WildFly domain controller and slave host controller processes take responsibility for ensuring each host has the needed content.

Unmanaged Deployments

An unmanaged deployment is one where the server directly deploys the content at a path you specify instead of making an internal copy and then deploying the copy.

Initially deploying an unmanaged deployment is much like deploying a managed one, except you tell WildFly that you do not want the deployment to be managed:

```
[standalone@localhost:9990 /] deploy ~/Desktop/test-application.war --unmanaged
'test-application.war' deployed successfully.
```

When you do this, instead of the server making a copy of the content at `/Desktop/test-application.war`, calculating the hash of the content, storing the hash in the configuration file and then installing the copy into the runtime, instead it will convert `/Desktop/test-application.war` to an absolute path, store the path in the configuration file, and then install the original content in the runtime.

You can also use unmanaged deployments in a domain:

```
[domain@localhost:9990 /] deploy /home/example/Desktop/test-application.war
--server-group=main-server-group --unmanaged
'test-application.war' deployed successfully.
```

However, before you run this command you must ensure that a copy of the content is present on all machines that have servers in the target server groups, all at the same filesystem path. The domain will not copy the file for you.

Undeploy is no different from a managed undeploy:

```
[standalone@localhost:9990 /] undeploy test-application.war
Successfully undeployed test-application.war.
```

Doing a replacement of the deployment with a new version is a bit different, the server is using the file you want to replace. You should undeploy the deployment, replace the content, and then deploy again. Or you can stop the server, replace the deployment and deploy again.

Deployment overlays

Deployment overlays are our way of 'overlaying' content into an existing deployment, without physically modifying the contents of the deployment archive. Possible use cases include swapping out deployment descriptors, modifying static web resources to change the branding of an application, or even replacing jar libraries with different versions.

Deployment overlays have a different lifecycle to a deployment. In order to use a deployment overlay, you first create the overlay, using the CLI or the management API. You then add files to the overlay, specifying the deployment paths you want them to overlay. Once you have created the overlay you then have to link it to a deployment name (which is done slightly differently depending on if you are in standalone or domain mode). Once you have created the link any deployment that matches the specified deployment name will have the overlay applied.

When you modify or create an overlay it will not affect existing deployments, they must be redeployed in order to take effect

Creating a deployment overlay

To create a deployment overlay the CLI provides a high level command to do all the steps specified above in one go. An example command is given below for both standalone and domain mode:

```
deployment-overlay add --name=myOverlay
--content=/WEB-INF/web.xml=/myFiles/myWeb.xml,/WEB-INF/ejb-jar.xml=/myFiles/myEjbJar.xml
--deployments=test.war,*-admin.war --redeploy-affected
```

```
deployment-overlay add --name=myOverlay
--content=/WEB-INF/web.xml=/myFiles/myWeb.xml,/WEB-INF/ejb-jar.xml=/myFiles/myEjbJar.xml
--deployments=test.war,*-admin.war --server-groups=main-server-group --redeploy-affected
```

5.19.9 Audit logging

WildFly comes with audit logging built in for management operations affecting the management model. By default it is turned off. The information is output as JSON records.

The default configuration of audit logging in standalone.xml looks as follows:

```
<management>
        <security-realms>
...
        </security-realms>
        <audit-log>
            <formatters>
                <json-formatter name="json-formatter"/>
            </formatters>
            <handlers>
                <file-handler name="file" formatter="json-formatter" path="audit-log.log"
relative-to="jboss.server.data.dir"/>
            </handlers>
            <logger log-boot="true" log-read-only="true" enabled="false">
                <handlers>
                    <handler name="file"/>
                </handlers>
            </logger>
        </audit-log>
...
```

Looking at this via the CLI it looks like

```
[standalone@localhost:9990 /]
/core-service=management/access=audit:read-resource(recursive=true)
{
    "outcome" => "success",
    "result" => {
        "file-handler" => {"file" => {
            "formatter" => "json-formatter",
            "max-failure-count" => 10,
            "path" => "audit-log.log",
            "relative-to" => "jboss.server.data.dir"
        }},
        "json-formatter" => {"json-formatter" => {
            "compact" => false,
            "date-format" => "yyyy-MM-dd HH:mm:ss",
            "date-separator" => " - ",
            "escape-control-characters" => false,
            "escape-new-line" => false,
            "include-date" => true
        }},
        "logger" => {"audit-log" => {
            "enabled" => false,
            "log-boot" => true,
            "log-read-only" => false,
            "handler" => {"file" => {}}
        }},
        "syslog-handler" => undefined
    }
}
```

To enable it via CLI you need just

```
[standalone@localhost:9990 /]
/core-service=management/access=audit/logger=audit-log:write-attribute(name=enabled,value=true)
{"outcome" => "success"}
```

Audit data are stored in standalone/data/audit-log.log.

The audit logging subsystem has a lot of internal dependencies, and it logs operations changing, enabling and disabling its components. When configuring or changing things at runtime it is a good idea to make these changes as part of a CLI batch. For example if you are adding a syslog handler you need to add the handler and its information as one step. Similarly if you are using a file handler, and want to change its `path` and `relative-to` attributes, that needs to happen as one step.

JSON Formatter

The first thing that needs configuring is the formatter, we currently support outputting log records as JSON. You can define several formatters, for use with different handlers. A log record has the following format, and it is the formatter's job to format the data presented:

```
2013-08-12 11:01:12 - {
    "type" : "core",
    "r/o" : false,
    "booting" : false,
    "version" : "8.0.0.Alpha4",
    "user" : "$local",
    "domainUUID" : null,
    "access" : "NATIVE",
    "remote-address" : "127.0.0.1/127.0.0.1",
    "success" : true,
    "ops" : [JMX|WFLY8:JMX subsystem configuration],
        "operation" : "write-attribute",
        "name" : "enabled",
        "value" : true,
        "operation-headers" : {"caller-type" : "user"}
    }]
}
```

It includes an optional timestamp and then the following information in the json record

Field name	Description
`type`	This can have the values `core`, meaning it is a management operation, or `jmx` meaning it comes from the jmx subsystem (see the jmx subsystem for configuration of the jmx subsystem's audit logging)
`r/o`	`true` if the operation does not change the management model, `false` otherwise
`booting`	`true` if the operation was executed during the bootup process, `false` if it was executed once the server is up and running
`version`	The version number of the WildFly instance
`user`	The username of the authenticated user. In this case the operation has been logged via the CLI on the same machine as the running server, so the special `$local` user is used
`domainUUID`	An ID to link together all operations as they are propagated from the Doman Controller to it servers, slave Host Controllers, and slave Host Controller servers
`access`	This can have one of the following values: *`NATIVE` - The operation came in through the native management interface, for example the CLI *`HTTP` - The operation came in through the domain HTTP interface, for example the admin console *`JMX` - The operation came in through the JMX subsystem. See JMX for how to configure audit logging for JMX.
`remote-address`	The address of the client executing this operation
`success`	`true` if the operation succeeded, `false` if it was rolled back
`ops`	The operations being executed. This is a list of the operations serialized to JSON. At boot this will be all the operations resulting from parsing the xml. Once booted the list will typically just contain a single entry

The json formatter resource has the following attributes:

Attribute	Description
`include-date`	Boolan toggling whether or not to include the timestamp in the formatted log records
`date-separator`	A string containing characters to separate the date and the rest of the formatted log message. Will be ignored if `include-date=false`
`date-format`	The date format to use for the timestamp as understood by `java.text.SimpleDateFormat`. Will be ignored if `include-date=false`
`compact`	If true will format the JSON on one line. There may still be values containing new lines, so if having the whole record on one line is important, set escape-new-line or escape-control-characters to true
`escape-control-characters`	If `true` it will escape all control characters (ascii entries with a decimal value < 32) with the ascii code in octal, e.g. a new line becomes '#012'. If this is `true`, it will override `escape-new-line=false`
`escape-new-line`	If `true` it will escape all new lines with the ascii code in octal, e.g. "#012".

Handlers

A handler is responsible for taking the formatted data and logging it to a location. There are currently two types of handlers, File and Syslog. You can configure several of each type of handler and use them to log information.

File handler

The file handlers log the audit log records to a file on the server. The attributes for the file handler are

Attribute	Description	Read Only
`formatter`	The name of a JSON formatter to use to format the log records	`false`
`path`	The path of the audit log file	`false`
`relative-to`	The name of another previously named path, or of one of the standard paths provided by the system. If `relative-to` is provided, the value of the `path` attribute is treated as relative to the path specified by this attribute	`false`
`failure-count`	The number of logging failures since the handler was initialized	`true`
`max-failure-count`	The maximum number of logging failures before disabling this handler	`false`
`disabled-due-to-failure`	`true` if this handler was disabled due to logging failures	`true`

In our standard configuration `path=audit-log.log` and `relative-to=jboss.server.data.dir`, typically this will be `$JBOSS_HOME/standalone/data/audit-log.log`

Syslog handler

The default configuration does not have syslog audit logging set up. Syslog is a better choice for audit logging since you can log to a remote syslog server, and secure the authentication to happen over TLS with client certificate authentication. Syslog servers vary a lot in their capabilities so not all settings in this section apply to all syslog servers. We have tested with rsyslog.

The address for the syslog handler is `/core-service=management/access=audit/syslog-handler=*` and just like file handlers you can add as many syslog entries as you like. The syslog handler resources reference the main RFC's for syslog a fair bit, for reference they can be found at:
*http://www.ietf.org/rfc/rfc3164.txt
*http://www.ietf.org/rfc/rfc5424.txt
*http://www.ietf.org/rfc/rfc6587.txt

The syslog handler resource has the following attributes:

`formatter`	The name of a JSON formatter to use to format the log records	`false`
`failure-count`	The number of logging failures since the handler was initialized	`true`
`max-failure-count`	The maximum number of logging failures before disabling this handler	`false`
`disabled-due-to-failure`	`true` if this handler was disabled due to logging failures	`true`
`syslog-format`	Whether to set the syslog format to the one specified in RFC-5424 or RFC-3164	`false`
`max-length`	The maximum length in bytes a log message, including the header, is allowed to be. If undefined, it will default to 1024 bytes if the syslog-format is RFC3164, or 2048 bytes if the syslog-format is RFC5424.	`false`
`truncate`	Whether or not a message, including the header, should truncate the message if the length in bytes is greater than the maximum length. If set to false messages will be split and sent with the same header values	`false`

When adding a syslog handler you also need to add the protocol it will use to communicate with the syslog server. The valid choices for protocol are `UDP`,`TCP` and `TLS`. The protocol must be added at the same time as you add the syslog handler, or it will fail. Also, you can only add one protocol for the handler.

UDP

Configures the handler to use UDP to communicate with the syslog server. The address of the `UDP` resource is `/core-service=management/access=audit/syslog-handler=*/protocol=udp`. The attributes of the `UDP` resource are:

Attribute	Description
`host`	The host of the syslog server for the udp requests
`port`	The port of the syslog server listening for the udp requests

TCP

Configures the handler to use TCP to communicate with the syslog server. The address of the `TCP` resource is `/core-service=management/access=audit/syslog-handler=*/protocol=tcp`. The attributes of the `TCP` resource are:

Attribute	Description
`host`	The host of the syslog server for the tcp requests
`port`	The port of the syslog server listening for the tcp requests
`message-transfer`	The message transfer setting as described in section 3.4 of RFC-6587. This can either be OCTET_COUNTING as described in section 3.4.1 of RFC-6587, or NON_TRANSPARENT_FRAMING as described in section 3.4.1 of RFC-6587

TLS

Configures the handler to use TLC to communicate securely with the syslog server. The address of the `TLS` resource is `/core-service=management/access=audit/syslog-handler=*/protocol=tls`. The attributes of the `TLS` resource are the same as for `TCP`:

Attribute	Description
`host`	The host of the syslog server for the tls requests
`port`	The port of the syslog server listening for the tls requests
`message-transfer`	The message transfer setting as described in section 3.4 of RFC-6587. This can either be OCTET_COUNTING as described in section 3.4.1 of RFC-6587, or NON_TRANSPARENT_FRAMING as described in section 3.4.1 of RFC-6587

If the syslog server's TLS certificate is not signed by a certificate signing authority, you will need to set up a truststore to trust the certificate. The resource for the trust store is a child of the `TLS` resource, and the full address is `/core-service=management/access=audit/syslog-handler=*/protocol=tls/authenticati`. The attributes of the truststore resource are:

Attribute	Description
`keystore-password`	The password for the truststore
`keystore-path`	The path of the truststore
`keystore-relative-to`	The name of another previously named path, or of one of the standard paths provided by the system. If `keystore-relative-to` is provided, the value of the `keystore-path` attribute is treated as relative to the path specified by this attribute

TLS with Client certificate authentication.
If you have set up the syslog server to require client certificate authentication, when creating your handler you will also need to set up a client certificate store containing the certificate to be presented to the syslog server. The address of the client certificate store resource is `/core-service=management/access=audit/syslog-handler=*/protocol=tls/authenticati` and its attributes are:

Attribute	Description
`keystore-password`	The password for the keystore
`key-password`	The password for the keystore key
`keystore-path`	The path of the keystore
`keystore-relative-to`	The name of another previously named path, or of one of the standard paths provided by the system. If `keystore-relative-to` is provided, the value of the `keystore-path` attribute is treated as relative to the path specified by this attribute

Logger configuration

The final part that needs configuring is the logger for the management operations. This references one or more handlers and is configured at `/core-service=management/access=audit/logger=audit-log`. The attributes for this resource are:

Attribute	Description
`enabled`	`true` to enable logging of the management operations
`log-boot`	`true` to log the management operations when booting the server, `false` otherwise
`log-read-only`	If `true` all operations will be audit logged, if `false` only operations that change the model will be logged

Then which handlers are used to log the management operations are configured as `handler=*` children of the logger.

Domain Mode (host specific configuration)

In domain mode audit logging is configured for each host in its `host.xml` file. This means that when connecting to the DC, the configuration of the audit logging is under the host's entry, e.g. here is the default configuration:

```
[domain@localhost:9990 /]
/host=master/core-service=management/access=audit:read-resource(recursive=true)
{
    "outcome" => "success",
    "result" => {
        "file-handler" => {
            "host-file" => {
                "formatter" => "json-formatter",
                "max-failure-count" => 10,
                "path" => "audit-log.log",
                "relative-to" => "jboss.domain.data.dir"
            },
            "server-file" => {
                "formatter" => "json-formatter",
                "max-failure-count" => 10,
                "path" => "audit-log.log",
                "relative-to" => "jboss.server.data.dir"
            }
        },
        "json-formatter" => {"json-formatter" => {
            "compact" => false,
            "date-format" => "yyyy-MM-dd HH:mm:ss",
            "date-separator" => " - ",
            "escape-control-characters" => false,
            "escape-new-line" => false,
            "include-date" => true
        }},
        "logger" => {"audit-log" => {
            "enabled" => false,
            "log-boot" => true,
            "log-read-only" => false,
            "handler" => {"host-file" => {}}
        }},
        "server-logger" => {"audit-log" => {
            "enabled" => false,
            "log-boot" => true,
            "log-read-only" => false,
            "handler" => {"server-file" => {}}
        }},
        "syslog-handler" => undefined
    }
}
```

We now have two file handlers, one called `host-file` used to configure the file to log management operations on the host, and one called `server-file` used to log management operations executed on the servers. Then `logger=audit-log` is used to configure the logger for the host controller, referencing the `host-file` handler. `server-logger=audit-log` is used to configure the logger for the managed servers, referencing the `server-file` handler. The attributes for `server-logger=audit-log` are the same as for `server-logger=audit-log` in the previous section. Having the host controller and server loggers configured independently means we can control audit logging for managed servers and the host controller independently.

5.19.10 Canceling Management Operations

WildFly includes the ability to use the CLI to cancel management requests that are not proceeding normally.

The cancel-non-progressing-operation operation

The `cancel-non-progressing-operation` operation instructs the target process to find any operation that isn't proceeding normally and cancel it.

On a standalone server:

```
[standalone@localhost:9990 /]
/core-service=management/service=management-operations:cancel-non-progressing-operation
{
    "outcome" => "success",
    "result" => "-1155777943"
}
```

The result value is an internal identification number for the operation that was cancelled.

On a managed domain host controller, the equivalent resource is in the host=<hostname> portion of the management resource tree:

```
[domain@localhost:9990 /]
/host=host-a/core-service=management/service=management-operations:cancel-non-progressing-operatio
"outcome" => "success",
    "result" => "2156877946"
}
```

An operation can be cancelled on an individual managed domain server as well:

```
[domain@localhost:9990 /]
/host=host-a/server=server-one/core-service=management/service=management-operations:cancel-non-pr
"outcome" => "success",
    "result" => "6497786512"
}
```

An operation is considered to be not proceeding normally if it has been executing with the exclusive operation lock held for longer than 15 seconds. Read-only operations do not acquire the exclusive operation lock, so this operation will not cancel read-only operations. Operations blocking waiting for another operation to release the exclusive lock will also not be cancelled.

If there isn't any operation that is failing to proceed normally, there will be a failure response:

```
[standalone@localhost:9990 /]
/core-service=management/service=management-operations:cancel-non-progressing-operation
{
    "outcome" => "failed",
    "failure-description" => "WFLYDM0089: No operation was found that has been holding the
operation execution write lock for long than [15] seconds",
    "rolled-back" => true
}
```

The find-non-progressing-operation operation

To simply learn the id of an operation that isn't proceeding normally, but not cancel it, use the `find-non-progressing-operation` operation:

```
[standalone@localhost:9990 /]
/core-service=management/service=management-operations:find-non-progressing-operation
{
    "outcome" => "success",
    "result" => "-1155777943"
}
```

If there is no non-progressing operation, the outcome will still be `success` but the result will be `undefined`.

Once the id of the operation is known, the management resource for the operation can be examined to learn more about its status.

Examining the status of an active operation

There is a management resource for any currently executing operation that can be queried:

```
[standalone@localhost:9990 /]
/core-service=management/service=management-operations/active-operation=-1155777943:read-resource(
"outcome" => "success",
    "result" => {
        "access-mechanism" => "undefined",
        "address" => [
            ("deployment" => "example")
        ],
        "caller-thread" => "management-handler-thread - 24",
        "cancelled" => false,
        "exclusive-running-time" => 101918273645L,
        "execution-status" => "awaiting-stability",
        "operation" => "deploy",
        "running-time" => 101918279999L
    }
}
```

The response includes the following attributes:

Field	Meaning
access-mechanism	The mechanism used to submit a request to the server. NATIVE, JMX, HTTP
address	The address of the resource targeted by the operation. The value in the final element of the address will be '<hidden>' if the caller is not authorized to address the operation's target resource.
caller-thread	The name of the thread that is executing the operation.
cancelled	Whether the operation has been cancelled.
exclusive-running-time	Amount of time in nanoseconds the operation has been executing with the exclusive operation execution lock held, or -1 if the operation does not hold the exclusive execution lock.
execution-status	The current activity of the operation. See below for details.
operation	The name of the operation, or '<hidden>' if the caller is not authorized to address the operation's target resource.
running-time	Amount of time the operation has been executing, in nanoseconds.

The following are the values for the `execution-status` attribute:

Value	Meaning
executing	The caller thread is actively executing
awaiting-other-operation	The caller thread is blocking waiting for another operation to release the exclusive execution lock
awaiting-stability	The caller thread has made changes to the service container and is waiting for the service container to stabilize
completing	The operation is committed and is completing execution
rolling-back	The operation is rolling back

All currently executing operations can be viewed in one request using the `read-children-resources` operation:

```
[standalone@localhost:9990 /]
/core-service=management/service=management-operations:read-children-resources(child-type=active-o
"outcome" => "success",
    "result" => {"-1155777943" => {
        "access-mechanism" => "undefined",
        "address" => [
            ("deployment" => "example")
        ],
        "caller-thread" => "management-handler-thread - 24",
        "cancelled" => false,
        "exclusive-running-time" => 101918273645L,
        "execution-status" => "awaiting-stability",
        "operation" => "deploy",
        "running-time" => 101918279999L
    },
    {"-1246693202" => {
        "access-mechanism" => "undefined",
        "address" => [
            ("core-service" => "management"),
            ("service" => "management-operations")
        ],
        "caller-thread" => "management-handler-thread - 30",
        "cancelled" => false,
        "exclusive-running-time" => -1L,
        "execution-status" => "executing",
        "operation" => "read-children-resources",
        "running-time" => 3356000L
    }}
}
```

Canceling a specific operation

The `cancel-non-progressing-operation` operation is a convenience operation for identifying and canceling an operation. However, an administrator can examine the active-operation resources to identify any operation, and then directly cancel it by invoking the `cancel` operation on the resource for the desired operation.

```
[standalone@localhost:9990 /]
/core-service=management/service=management-operations/active-operation=-1155777943:cancel
{
    "outcome" => "success",
    "result" => undefined
}
```

Controlling operation blocking time

As an operation executes, the execution thread may block at various points, particularly while waiting for the service container to stabilize following any changes. Since an operation may be holding the exclusive execution lock while blocking, in WildFly execution behavior was changed to ensure that blocking will eventually time out, resulting in roll back of the operation.

The default blocking timeout is 300 seconds. This is intentionally long, as the idea is to only trigger a timeout when something has definitely gone wrong with the operation, without any false positives.

An administrator can control the blocking timeout for an individual operation by using the `blocking-timeout` operation header. For example, if a particular deployment is known to take an extremely long time to deploy, the default 300 second timeout could be increased:

```
[standalone@localhost:9990 /] deploy /tmp/mega.war --headers={blocking-timeout=450}
```

Note the blocking timeout is **not** a guaranteed maximum execution time for an operation. If it only a timeout that will be enforced at various points during operation execution.

5.19.11 Command line parameters

To start up a WildFly managed domain, execute the `$JBOSS_HOME/bin/domain.sh` script. To start up a standalone server, execute the `$JBOSS_HOME/bin/standalone.sh`. With no arguments, the default configuration is used. You can override the default configuration by providing arguments on the command line, or in your calling script.

System properties

To set a system property, pass its new value using the standard jvm `-Dkey=value` options:

```
$JBOSS_HOME/bin/standalone.sh -Djboss.home.dir=some/location/wildFly \
    -Djboss.server.config.dir=some/location/wildFly/custom-standalone
```

This command starts up a standalone server instance using a non-standard AS home directory and a custom configuration directory. For specific information about system properties, refer to the definitions below.

Instead of passing the parameters directly, you can put them into a properties file, and pass the properties file to the script, as in the two examples below.

```
$JBOSS_HOME/bin/domain.sh --properties=/some/location/jboss.properties
$JBOSS_HOME/bin/domain.sh -P=/some/location/jboss.properties
```

Note however, that properties set this way are not processed as part of JVM launch. They are processed early in the boot process, but this mechanism should not be used for setting properties that control JVM behavior (e.g. java.net.perferIPv4Stack) or the behavior of the JBoss Modules classloading system.

The syntax for passing in parameters and properties files is the same regardless of whether you are running the `domain.sh`, `standalone.sh`, or the Microsoft Windows scripts`domain.bat` or `standalone.bat`.

The properties file is a standard Java property file containing `key=value` pairs:

```
jboss.home.dir=/some/location/wildFly
jboss.domain.config.dir=/some/location/wildFly/custom-domain
```

System properties can also be set via the xml configuration files. Note however that for a standalone server properties set this way will not be set until the xml configuration is parsed and the commands created by the parser have been executed. So this mechanism should not be used for setting properties whose value needs to be set before this point.

Controlling filesystem locations with system properties

The standalone and the managed domain modes each use a default configuration which expects various files and writable directories to exist in standard locations. Each of these standard locations is associated with a system property, which has a default value. To override a system property, pass its new value using the one of the mechanisms above. The locations which can be controlled via system property are:

Standalone

Property name	Usage	Default value
`java.ext.dirs`	The JDK extension directory paths	`null`
`jboss.home.dir`	The root directory of the WildFly installation.	Set by `standalone.sh` to `$JBOSS_HOME`
`jboss.server.base.dir`	The base directory for server content.	`jboss.home.dir`/standalone
`jboss.server.config.dir`	The base configuration directory.	`jboss.server.base.dir` /configuration
`jboss.server.data.dir`	The directory used for persistent data file storage.	`jboss.server.base.dir` /data
`jboss.server.log.dir`	The directory containing the `server.log` file.	`jboss.server.base.dir`/log
`jboss.server.temp.dir`	The directory used for temporary file storage.	`jboss.server.base.dir`/tmp
`jboss.server.deploy.dir`	The directory used to store deployed content	`jboss.server.data.dir` /content

Managed Domain

Property name	Usage	Default value
`jboss.home.dir`	The root directory of the WildFly installation.	Set by `domain.sh` to `$JBOSS_HOME`
`jboss.domain.base.dir`	The base directory for domain content.	`jboss.home.dir`/domain
`jboss.domain.config.dir`	The base configuration directory	`jboss.domain.base.dir`/configuration
`jboss.domain.data.dir`	The directory used for persistent data file storage.	`jboss.domain.base.dir`/data
`jboss.domain.log.dir`	The directory containing the `host-controller.log` and `process-controller.log` files	`jboss.domain.base.dir`/log
`jboss.domain.temp.dir`	The directory used for temporary file storage	`jboss.domain.base.dir`/tmp
`jboss.domain.deployment.dir`	The directory used to store deployed content	`jboss.domain.base.dir`/content
`jboss.domain.servers.dir`	The directory containing the output for the managed server instances	`jboss.domain.base.dir`/servers

Other command line parameters

The first acceptable format for command line arguments to the WildFly launch scripts is

```
--name=value
```

For example:

```
$JBOSS_HOME/bin/standalone.sh --server-config=standalone-ha.xml
```

If the parameter name is a single character, it is prefixed by a single '-' instead of two. Some parameters have both a long and short option.

```
-x=value
```

For example:

```
$JBOSS_HOME/bin/standalone.sh -P=/some/location/jboss.properties
```

For some command line arguments frequently used in previous major releases of WildFly, replacing the "=" in the above examples with a space is supported, for compatibility.

```
-b 192.168.100.10
```

If possible, use the `-x=value` syntax. New parameters will always support this syntax.

The sections below describe the command line parameter names that are available in standalone and domain mode.

Standalone

Name	Default if absent	Value
`--admin-only`	-	Set the server's running type to ADMIN_ONLY causing it to open administrative interfaces and accept management requests but not start other runtime services or accept end user requests.
`--server-config` `-c`	standalone.xml	A relative path which is interpreted to be relative to `jboss.server.config.dir`. The name of the configuration file to use.
`--read-only-server-config`	-	A relative path which is interpreted to be relative to `jboss.server.config.dir`. This is similar to `--server-config` but if this alternative is specified the server will not overwrite the file when the management model is changed. However a full versioned history is maintained of the file.

Managed Domain

Name	Default if absent	Value
`--admin-only`	-	Set the server's running type to ADMIN_ONLY causing it to open administrative interfaces and accept management requests but not start servers or, if this host controller is the master for the domain, accept incoming connections from slave host controllers.
`--domain-config` `-c`	domain.xml	A relative path which is interpreted to be relative to `jboss.domain.config.dir`. The name of the domain wide configuration file to use.
`--read-only-domain-config`	-	A relative path which is interpreted to be relative to `jboss.domain.config.dir`. This is similar to `--domain-config` but if this alternative is specified the host controller will not overwrite the file when the management model is changed. However a full versioned history is maintained of the file.
`--host-config`	host.xml	A relative path which is interpreted to be relative to `jboss.domain.config.dir`. The name of the host-specific configuration file to use.
`--read-only-host-config`	-	A relative path which is interpreted to be relative to `jboss.domain.config.dir`. This is similar to `--host-config` but if this alternative is specified the host controller will not overwrite the file when the management model is changed. However a full versioned history is maintained of the file.

The following parameters take no value and are only usable on slave host controllers (i.e. hosts configured to connect to a `remote` domain controller.)

Name	Function
`--backup`	Causes the slave host controller to create and maintain a local copy (domain.cached-remote.xml) of the domain configuration. If *ignore-unused-configuration* is unset in host.xml, a complete copy of the domain configuration will be stored locally, otherwise the configured value of *ignore-unused-configuration* in host.xml will be used. (See ignore-unused-configuration for more details.)
`--cached-dc`	If the slave host controller is unable to contact the master domain controller to get its configuration at boot, this option will allow the slave host controller to boot and become operational using a previously cached copy of the domain configuration (domain.cached-remote.xml.) If the cached configuration is not present, this boot will fail. This file is created using using one of the following methods: - A previously successful connection to the master domain controller using --backup or --cached-dc. - Copying the domain configuration from an alternative host to domain/configuration/domain.cached-remote.xml. The unavailable master domain controller will be polled periodically for availability, and once becoming available, the slave host controller will reconnect to the master host controller and synchronize the domain configuration. During the interval the master domain controller is unavailable, the slave host controller will not be able make any modifications to the domain configuration, but it may launch servers and handle requests to deployed applications etc.

Common parameters

These parameters apply in both standalone or managed domain mode:

Name	Function
`-b=<value>`	Sets system property `jboss.bind.address` to <value>. See Controlling the Bind Address with -b for further details.
`-b<name>=<value>`	Sets system property `jboss.bind.address.`<name> to <value> where *name* can vary. See Controlling the Bind Address with -b for further details.
`-u=<value>`	Sets system property `jboss.default.multicast.address` to <value>. See Controlling the Default Multicast Address with -u for further details.

`--version`
`-v`
`-V`
	Prints the version of WildFly to standard output and exits the JVM.

`--help` `-h`	Prints a help message explaining the options and exits the JVM.

Controlling the Bind Address with -b

WildFly binds sockets to the IP addresses and interfaces contained in the `<interfaces>` elements in `standalone.xml`, `domain.xml` and `host.xml`. (See Interfaces and Socket Bindings for further information on these elements.) The standard configurations that ship with WildFly includes two interface configurations:

```
<interfaces>
    <interface name="management">
        <inet-address value="${jboss.bind.address.management:127.0.0.1}"/>
    </interface>
    <interface name="public">
       <inet-address value="${jboss.bind.address:127.0.0.1}"/>
    </interface>
</interfaces>
```

Those configurations use the values of system properties `jboss.bind.address.management` and `jboss.bind.address` if they are set. If they are not set, 127.0.0.1 is used for each value.

As noted in Common Parameters, the AS supports the `-b` and `-b<name>` command line switches. The only function of these switches is to set system properties `jboss.bind.address` and `jboss.bind.address.`<name> respectively. However, because of the way the standard WildFly configuration files are set up, using the `-b` switches can indirectly control how the AS binds sockets.

If your interface configurations match those shown above, using this as your launch command causes all sockets associated with interface named "public" to be bound to `192.168.100.10`.

```
$JBOSS_HOME/bin/standalone.sh -b=192.168.100.10
```

In the standard config files, public interfaces are those not associated with server management. Public interfaces handle normal end-user requests.

Interface names

The interface named "public" is not inherently special. It is provided as a convenience. You can name your interfaces to suit your environment.

To bind the public interfaces to all IPv4 addresses (the IPv4 wildcard address), use the following syntax:

```
$JBOSS_HOME/bin/standalone.sh -b=0.0.0.0
```

You can also bind the management interfaces, as follows:

```
$JBOSS_HOME/bin/standalone.sh -bmanagement=192.168.100.10
```

In the standard config files, management interfaces are those sockets associated with server management, such as the socket used by the CLI, the HTTP socket used by the admin console, and the JMX connector socket.

Be Careful

The `-b` switch only controls the interface bindings because the standard config files that ship with WildFly sets things up that way. If you change the `<interfaces>` section in your configuration to no longer use the system properties controlled by `-b`, then setting `-b` in your launch command will have no effect.

For example, this perfectly valid setting for the "public" interface causes `-b` to have no effect on the "public" interface:

```
<interface name="public">
   <nic name="eth0"/>
</interface>
```

The key point is **the contents of the configuration files determine the configuration. Settings like `-b` are not overrides of the configuration files.** They only provide a shorter syntax for setting a system properties that may or may not be referenced in the configuration files. They are provided as a convenience, and you can choose to modify your configuration to ignore them.

Controlling the Default Multicast Address with -u

WildFly may use multicast communication for some services, particularly those involving high availability clustering. The multicast addresses and ports used are configured using the `socket-binding` elements in `standalone.xml` and `domain.xml`. (See Socket Bindings for further information on these elements.) The standard HA configurations that ship with WildFly include two socket binding configurations that use a default multicast address:

```
<socket-binding name="jgroups-mping" port="0"
multicast-address="${jboss.default.multicast.address:230.0.0.4}" multicast-port="45700"/>
<socket-binding name="jgroups-udp" port="55200"
multicast-address="${jboss.default.multicast.address:230.0.0.4}" multicast-port="45688"/>
```

Those configurations use the values of system property `jboss.default.multicast.address` if it is set. If it is not set, 230.0.0.4 is used for each value. (The configuration may include other socket bindings for multicast-based services that are not meant to use the default multicast address; e.g. a binding the mod-cluster services use to communicate on a separate address/port with Apache httpd servers.)

As noted in Common Parameters, the AS supports the `-u` command line switch. The only function of this switch is to set system property `jboss.default.multicast.address`. However, because of the way the standard AS configuration files are set up, using the `-u` switches can indirectly control how the AS uses multicast.

If your socket binding configurations match those shown above, using this as your launch command causes the service using those sockets configurations to be communicate over multicast address `230.0.1.2`.

```
$JBOSS_HOME/bin/standalone.sh -u=230.0.1.2
```

Be Careful

As with the `-b` switch, the `-u` switch only controls the multicast address used because the standard config files that ship with WildFly sets things up that way. If you change the `<socket-binding>` sections in your configuration to no longer use the system properties controlled by `-u`, then setting `-u` in your launch command will have no effect.

5.19.12 Configuration file history

The management operations may modify the model. When this occurs the xml backing the model is written out again reflecting the latest changes. In addition a full history of the file is maintained. The history of the file goes in a separate directory under the configuration directory.

As mentioned in Command line parameters#parameters the default configuration file can be selected using a command-line parameter. For a standalone server instance the history of the active `standalone.xml` is kept in `jboss.server.config.dir`/standalone_xml_history (See Command line parameters#standalone_system_properties for more details). For a domain the active `domain.xml` and `host.xml` histories are kept in `jboss.domain.config.dir`/domain_xml_history and `jboss.domain.config.dir`/host_xml_history.

The rest of this section will only discuss the history for `standalone.xml`. The concepts are exactly the same for `domain.xml` and `host.xml`.

Within `standalone_xml_history` itself following a successful first time boot we end up with three new files:

- `standalone.initial.xml` - This contains the original configuration that was used the first time we successfully booted. This file will never be overwritten. You may of course delete the history directory and any files in it at any stage.
- `standalone.boot.xml` - This contains the original configuration that was used for the last successful boot of the server. This gets overwritten every time we boot the server successfully.
- `standalone.last.xml` - At this stage the contents will be identical to `standalone.boot.xml`. This file gets overwritten each time the server successfully writes the configuration, if there was an unexpected failure writing the configuration this file is the last known successful write.

`standalone_xml_history` contains a directory called `current` which should be empty. Now if we execute a management operation that modifies the model, for example adding a new system property using the CLI:

```
[standalone@localhost:9990 /] /system-property=test:add(value="test123")
{"outcome" => "success"}
```

What happens is:

- The original configuration file is backed up to `standalone_xml_history/current/standalone.v1.xml`. The next change to the model would result in a file called `standalone.v2.xml` etc. The 100 most recent of these files are kept.
- The change is applied to the original configuration file
- The changed original configuration file is copied to `standalone.last.xml`

When restarting the server, any existing `standalone_xml_history/current` directory is moved to a new timestamped folder within the `standalone_xml_history`, and a new `current` folder is created. These timestamped folders are kept for 30 days.

Snapshots

In addition to the backups taken by the server as described above you can manually take take snapshots which will be stored in the `snapshot` folder under the `_xml_history` folder, the automatic backups described above are subject to automatic house keeping so will eventually be automatically removed, the snapshots on the other hand can be entirely managed by the administrator.

You may also take your own snapshots using the CLI:

```
[standalone@localhost:9990 /] :take-snapshot
{
    "outcome" => "success",
    "result" => {"name" =>
"/Users/kabir/wildfly/standalone/configuration/standalone_xml_history/snapshot/20110630-172258657s
```

You can also use the CLI to list all the snapshots

```
[standalone@localhost:9990 /] :list-snapshots
{
    "outcome" => "success",
    "result" => {
        "directory" =>
"/Users/kabir/wildfly/standalone/configuration/standalone_xml_history/snapshot",
        "names" => [
            "20110630-165714239standalone.xml",
            "20110630-165821795standalone.xml",
            "20110630-170113581standalone.xml",
            "20110630-171411463standalone.xml",
            "20110630-171908397standalone.xml",
            "20110630-172258657standalone.xml"
        ]
    }
}
```

To delete a particular snapshot:

```
[standalone@localhost:9990 /] :delete-snapshot(name="20110630-165714239standalone.xml")
{"outcome" => "success"}
```

and to delete all snapshots:

```
[standalone@localhost:9990 /] :delete-snapshot(name="all")
{"outcome" => "success"}
```

In domain mode executing the snapshot operations against the root node will work against the domain model. To do this for a host model you need to navigate to the host in question:

```
[domain@localhost:9990 /] /host=master:list-snapshots
{
    "outcome" => "success",
    "result" => {
        "domain-results" => {"step-1" => {
            "directory" =>
"/Users/kabir/wildfly/domain/configuration/host_xml_history/snapshot",
            "names" => [
                "20110630-141129571host.xml",
                "20110630-172522225host.xml"
            ]
        }},
        "server-operations" => undefined
    }
}
```

Subsequent Starts

For subsequent server starts it may be desirable to take the state of the server back to one of the previously known states, for a number of items an abbreviated reverence to the file can be used:

Abreviation	Parameter	Description
`initial`	`--server-config=initial`	This will start the server using the initial configuration first used to start the server.
`boot`	`--server-config=boot`	This will use the configuration from the last successful boot of the server.
`last`	`--server-config=last`	This will start the server using the configuration backed up from the last successful save.
`v?`	`--server-config=v?`	This will server the _xml_history/current folder for the configuration where ? is the number of the backup to use.
`-?`	`--server-config=-?`	The server will be started after searching the snapshot folder for the configuration which matches this prefix.

In addition to this the `--server-config` parameter can always be used to specify a configuration relative to the `jboss.server.config.dir` and finally if no matching configuration is found an attempt to locate the configuration as an absolute path will be made.

5.19.13 Deployment Overlays

Deployment overlays are our way of 'overlaying' content into an existing deployment, without physically modifying the contents of the deployment archive. Possible use cases include swapping out deployment descriptors, modifying static web resources to change the branding of an application, or even replacing jar libraries with different versions.

Deployment overlays have a different lifecycle to a deployment. In order to use a deployment overlay, you first create the overlay, using the CLI or the management API. You then add files to the overlay, specifying the deployment paths you want them to overlay. Once you have created the overlay you then have to link it to a deployment name (which is done slightly differently depending on if you are in standalone or domain mode). Once you have created the link any deployment that matches the specified deployment name will have the overlay applied.

When you modify or create an overlay it will not affect existing deployments, they must be redeployed in order to take effect

Creating a deployment overlay

To create a deployment overlay the CLI provides a high level command to do all the steps specified above in one go. An example command is given below for both standalone and domain mode:

```
deployment-overlay add --name=myOverlay
--content=/WEB-INF/web.xml=/myFiles/myWeb.xml,/WEB-INF/ejb-jar.xml=/myFiles/myEjbJar.xml
--deployments=test.war,*-admin.war --redeploy-affected
```

```
deployment-overlay add --name=myOverlay
--content=/WEB-INF/web.xml=/myFiles/myWeb.xml,/WEB-INF/ejb-jar.xml=/myFiles/myEjbJar.xml
--deployments=test.war,*-admin.war --server-groups=main-server-group --redeploy-affected
```

5.19.14 JVM settings

Configuration of the JVM settings is different for a managed domain and a standalone server. In a managed domain, the domain controller components are responsible for starting and stoping server processes and hence determine the JVM settings. For a standalone server, it's the responsibility of the process that started the server (e.g. passing them as command line arguments).

Managed Domain

In a managed domain the JVM settings can be declared at different scopes: For a specific server group, for a host or for a particular server. If not declared, the settings are inherited from the parent scope. This allows you to customize or extend the JVM settings within every layer.

Let's take a look at the JVM declaration for a server group:

```
<server-groups>
       <server-group name="main-server-group" profile="default">
           <jvm name="default">
               <heap size="64m" max-size="512m"/>
           </jvm>
           <socket-binding-group ref="standard-sockets"/>
       </server-group>
       <server-group name="other-server-group" profile="default">
           <jvm name="default">
               <heap size="64m" max-size="512m"/>
           </jvm>
           <socket-binding-group ref="standard-sockets"/>
       </server-group>
</server-groups>
```

(See `domain/configuration/domain.xml`)

In this example the server group "main-server-group" declares a heap size of `64m` and a maximum heap size of `512m`. Any server that belongs to this group will inherit these settings. You can change these settings for the group as a whole, or a specific server or host:

```
<servers>
       <server name="server-one" group="main-server-group" auto-start="true">
           <jvm name="default"/>
       </server>
       <server name="server-two" group="main-server-group" auto-start="true">
           <jvm name="default">
               <heap size="64m" max-size="256m"/>
           </jvm>
           <socket-binding-group ref="standard-sockets" port-offset="150"/>
       </server>
       <server name="server-three" group="other-server-group" auto-start="false">
           <socket-binding-group ref="standard-sockets" port-offset="250"/>
       </server>
</servers>
```

(See domain/configuration/host.xml)

In this case, *server-two*, belongs to the *main-server-group* and inherits the JVM settings named *default*, but declares a lower maximum heap size.

```
[domain@localhost:9999 /] /host=local/server-config=server-two/jvm=default:read-resource
{
   "outcome" => "success",
   "result" => {
       "heap-size" => "64m",
       "max-heap-size" => "256m",
   }
}
```

Standalone Server

For a standalone sever you have to pass in the JVM settings either as command line arguments when executing the `$JBOSS_HOME/bin/standalone.sh` script, or by declaring them in `$JBOSS_HOME/bin/standalone.conf`. (For Windows users, the script to execute is `%JBOSS_HOME%/bin/standalone.bat` while the JVM settings can be declared in `%JBOSS_HOME%/bin/standalone.conf.bat`.)

5.19.15 Starting & stopping Servers in a Managed Domain

Starting a standalone server is done through the `bin/standalone.sh` script. However in a managed domain server instances are managed by the domain controller and need to be started through the management layer:

First of all, get to know which `servers` are configured on a particular `host`:

```
[domain@localhost:9990 /] :read-children-names(child-type=host)
{
   "outcome" => "success",
   "result" => ["local"]
}


[domain@localhost:9990 /] /host=local:read-children-names(child-type=server-config)
{
   "outcome" => "success",
   "result" => [
       "my-server",
       "server-one",
       "server-three"
   ]
}
```

Now that we know, that there are two `servers` configured on `host` "*local*", we can go ahead and check their status:

```
[domain@localhost:9990 /]
/host=local/server-config=server-one:read-resource(include-runtime=true)
{
   "outcome" => "success",
   "result" => {
       "auto-start" => true,
       "group" => "main-server-group",
       "interface" => undefined,
       "name" => "server-one",
       "path" => undefined,
       "socket-binding-group" => undefined,
       "socket-binding-port-offset" => undefined,
       "status" => "STARTED",
       "system-property" => undefined,
       "jvm" => {"default" => undefined}
   }
}
```

You can change the server state through the "*start*" and "*stop*" operations

```
[domain@localhost:9990 /] /host=local/server-config=server-one:stop
{
   "outcome" => "success",
   "result" => "STOPPING"
}
```

Navigating through the domain topology is much more simple when you use the web interface.

5.19.16 Suspend, Resume and Graceful shutdown

Core Concepts

Wildfly introduces the ability to suspend and resume servers. This can be combined with shutdown to enable the server to gracefully finish processing all active requests and then shut down. When a server is suspended it will immediately stop accepting new requests, but wait for existing request to complete. A suspended server can be resumed at any point, and will begin processing requests immediately. Suspending and resuming has no effect on deployment state (e.g. if a server is suspended singleton EJB's will not be destroyed). As of Wildfly 11 it is also possible to start a server in suspended mode which means it will not accept requests until it has been resumed, servers will also be suspended during the boot process, so no requests will be accepted until the startup process is 100% complete.

Suspend/Resume has no effect on management operations, management operations can still be performed while a server is suspended. If you wish to perform a management operation that will affect the operation of the server (e.g. changing a datasource) you can suspend the server, perform the operation, then resume the server. This allows all requests to finish, and makes sure that no requests are running while the management changes are taking place.

When a server is suspending it goes through four different phases:

- **RUNNING** - The normal state, the server is accepting requests and running normally
- **PRE_SUSPEND** - In PRE_SUSPEND the server will notify external parties that it is about to suspend, for example mod_cluster will notify the load balancer that the deployment is suspending. Requests are still accepted in this phase.
- **SUSPENDING** - All new requests are rejected, and the server is waiting for all active requests to finish. If there are no active requests at suspend time this phase will be skipped.
- **SUSPENDED** - All requests have completed, and the server is suspended.

Starting Suspended

In order to start into suspended mode when using a standalone server you need to add **--start-mode=suspend** to the command line. It is also possible to specify the start-mode in the **reload** operation to cause the server to reload into suspended mode (other possible values for start-mode are **normal** and **admin-only**).

In domain mode servers can be started in suspended mode by passing the **suspend=true** parameter to any command that causes a server to start, restart or reload (e.g. :start-servers(suspend=true)).

The Request Controller Subsystem

Wildfly introduces a new subsystem called the Request Controller Subsystem. This optional subsystem tracks all requests at their entry point, which how the graceful shutdown mechanism know when all requests are done (it also allows you to provide a global limit on the total number of running requests).

If this subsystem is not present suspend/resume will be limited, in general things that happen in the PRE_SUSPEND phase will work as normal (stopping message delivery, notifying the load balancer), however the server will not wait for all requests to complete and instead move straight to SUSPENDED mode.

There is a small performance penalty associated with the request controller subsystem (about on par with enabling statistics), so if you do not require the suspend/resume functionality this subsystem can be removed to get a small performance boost.

Subsystem Integrations

Suspend/Resume is a service provided by the Wildfly platform that any subsystem may choose to integrate with. Some subsystems integrate directly with the suspend controller, while others integrate through the request controller subsystem.

The following subsystems support graceful shutdown. Note that only subsystems that provide an external entry point to the server need graceful shutdown support, for example the JAX-RS subsystem does not require suspend/resume support as all access to JAX-RS is through the web connector.

- **Undertow** - Undertow will wait for all requests to finish
- **mod_cluster** - The mod_cluster subsystem will notify the load balancer that the server is suspending in the PRE_SUSPEND phase.
- **EJB** - EJB will wait for all remote EJB requests and MDB message deliveries to finish. Delivery to MDB's is stopped in the PRE_SUSPEND phase. EJB timers are suspended, and missed timers will be activated when the server is resumed.
- **Batch** - Batch jobs will be stopped at a checkpoint while the server is suspending. They will be restarted from that checkpoint when the server returns to running mode.
- **EE Concurrency** - The server will wait for all active jobs to finish. All jobs that have already been queued will be skipped.
- **Transactions** - transaction subsystem waits for all running transactions to finish while server is suspending. During that time server refuses to start any new transaction. But any in-flight transaction will be serviced - e.g. it means that server accepts any incoming remote call which carries context of the transaction already started at the suspending server.
  When you work with EJBs you have to enable the graceful shutdown functionality by setting attribute `enable-graceful-txn-shutdown` to `true`.
  (at the `ejb3 subsystem` xml, for example):
  `<enable-graceful-txn-shutdown value="false"/>`
  By **default** graceful shutdown it's **disabled** for ejb subsystem.
  The reason is that the behavior might be unwelcome in cluster environments, as the server notifies remote clients that the node is no longer available for remote calls only after the transactions are finished. During that brief window of time, the client of a cluster may send a new request to a node that is shutting down and will refuse the request because it is not related to an existing transaction.
  If this attribute `enable-graceful-txn-shutdown` is set to `false`, we disable the graceful behavior and EJB clients will not attempt to invoke the node when it suspends, regardless of active transactions.

Standalone Mode

Suspend/Resume can be controlled via the following CLI operations in standalone mode:

```
:suspend(timeout=z)
```

Suspends the server. If the timeout is specified it will wait up to the specified number of seconds for all requests to finish. If there is no timeout specified or the value is less than zero it will wait indefinitely.

```
:resume
```

Resumes a previously suspended server. The server should be able to begin serving requests immediately.

```
:read-attribute(name=suspend-state)
```

Returns the current suspend state of the server.

```
:shutdown(timeout=x)
```

If a timeout parameter is passed to the shutdown command then a graceful shutdown will be performed. The server will be suspended, and will wait up to the specified number of seconds for all requests to finish before shutting down. A timeout value of less than zero means it will wait indefinitely.

Domain Mode

Domain mode has similar commands as standalone mode, however they can be applied at both the global and server group levels:

Whole Domain

```
:suspend-servers(timeout=x)
```

:resume-servers

:stop-servers(timeout=x)

Server Group

```
/server-group=main-server-group:suspend-servers(timeout=x)
```

```
/server-group=main-server-group:resume-servers
```

```
/server-group=main-server-group:stop-servers(timeout=x)
```

Server

```
/host=master/server-config=server-one:suspend(timeout=x)
```

/host=master/server-config=server-one:resume

/host=master/server-config=server-one:stop(timeout=x)

5.20 Authorizing management actions with Role Based Access Control

WildFly introduces a Role Based Access Control scheme that allows different administrative users to have different sets of permissions to read and update parts of the management tree. This replaces the simple permission scheme used in JBoss AS 7, where anyone who could successfully authenticate to the management security realm would have all permissions.

5.20.1 Access Control Providers

WildFly ships with two access control "providers", the "simple" provider, and the "rbac" provider. The "simple" provider is the default, and provides a permission scheme equivalent to the JBoss AS 7 behavior where any authenticated administrator has all permissions. The "rbac" provider gives the finer grained permission scheme that is the focus of this section.

The access control configuration is included in the management section of a standalone server's standalone.xml, or in a new "management" section in a managed domain's domain.xml. The access control policy is centrally configured in a managed domain.

```
<management>
    . . .
    <access-control provider="simple">
        <role-mapping>
            <role name="SuperUser">
                <include>
                    <user name="$local"/>
                </include>
            </role>
        </role-mapping>
    </access-control>
</management>
```

As you can see, the provider is set to "simple" by default. With the "simple" provider, the nested "role-mapping" section is not actually relevant. It's there to help ensure that if the provider attribute is switched to "rbac" there will be at least one user mapped to a role that can continue to administer the system. This default mapping assigns the "$local" user name to the RBAC role that provides all permissions, the "SuperUser" role. The "$local" user name is the name an administrator will be assigned if he or she uses the CLI on the same system as the WildFly instance and the "local" authentication scheme is enabled.

5.20.2 RBAC provider overview

The access control scheme implemented by the "rbac" provider is based on seven standard roles. A role is a named set of permissions to perform one of the actions: addressing (i.e. looking up) a management resource, reading it, or modifying it. The different roles have constraints applied to their permissions that are used to determine whether the permission is granted.

RBAC roles

The seven standard roles are divided into two broad categories, based on whether the role can deal with items that are considered to be "security sensitive". Resources, attributes and operations that may affect administrative security (e.g. security realm resources and attributes that contain passwords) are "security sensitive".

Four roles are not given permissions for "security sensitive" items:

- Monitor – a read-only role. Cannot modify any resource.
- Operator – Monitor permissions, plus can modify runtime state, but cannot modify anything that ends up in the persistent configuration. Could, for example, restart a server.
- Maintainer – Operator permissions, plus can modify the persistent configuration.
- Deployer – like a Maintainer, but with permission to modify persistent configuration constrained to resources that are considered to be "application resources". A deployment is an application resource. The messaging server is not. Items like datasources and JMS destinations are not considered to be application resources by default, but this is configurable.

Three roles are granted permissions for security sensitive items:

- SuperUser – has all permissions. Equivalent to a JBoss AS 7 administrator.
- Administrator – has all permissions except cannot read or write resources related to the administrative audit logging system.
- Auditor – can read anything. Can only modify the resources related to the administrative audit logging system.

The Auditor and Administrator roles are meant for organizations that want a separation of responsibilities between those who audit normal administrative actions and those who perform them, with those who perform most actions (Administrator role) not being able to read or alter the auditing configuration.

Access control constraints

The following factors are used to determine whether a given role is granted a permission:

- What the requested action is (address, read, write)
- Whether the resource, attribute or operation affects the persistent configuration
- Whether the resource, attribute or operation is related to the administrative audit logging function
- Whether the resource, attribute or operation is configured as security sensitive
- Whether an attribute or operation parameter value has a security vault expression
- Whether a resource is considered to be associated with applications, as opposed to being part of a general container configuration

The first three of these factors are non-configurable; the latter three allow some customization. See "Configuring constraints" for details.

Addressing a resource

As mentioned above, permissions are granted to perform one of three actions, addressing a resource, reading it, and modifying. The latter two actions are fairly self-explanatory. But what is meant by "addressing" a resource?

"Addressing" a resource refers to taking an action that allows the user to determine whether a resource at a given address actually exists. For example, the "read-children-names" operation lets a user determine valid addresses. Trying to read a resource and getting a "Permission denied" error also gives the user a clue that there actually is a resource at the requested address.

Some resources may include sensitive information as part of their address. For example, security realm resources include the realm name as the last element in the address. That realm name is potentially security sensitive; for example it is part of the data used when creating a hash of a user password. Because some addresses may contain security sensitive data, a user needs permission to even "address" a resource. If a user attempts to address a resource and does not have permission, they will not receive a "permission denied" type error. Rather, the system will respond as if the resource does not even exist, e.g. excluding the resource from the result of the "read-children-names" operation or responding with a "No such resource" error instead of "Permission denied" if the user is attempting to read or write the resource.

Another term for "addressing" a resource is "looking up" the resource.

5.20.3 Switching to the "rbac" provider

Use the CLI to switch the access-control provider.

Before changing the provider to "rbac", be sure your configuration has a user who will be mapped to one of the RBAC roles, preferably with at least one in the Administrator or SuperUser role. Otherwise your installation will not be manageable except by shutting it down and editing the xml configuration. If you have started with one of the standard xml configurations shipped with WildFly, the "$local" user will be mapped to the "SuperUser" role and the "local" authentication scheme will be enabled. This will allow a user running the CLI on the same system as the WildFly process to have full administrative permissions. Remote CLI users and web-based admin console users will have no permissions.

We recommend mapping at least one user besides "$local" before switching the provider to "rbac". You can do all of the configuration associated with the "rbac" provider even when the provider is set to "simple"

The management resources related to access control are located in the `core-service=management/access=authorization` portion of the management resource tree. Update the `provider` attribute to change between the "simple" and "rbac" providers. Any update requires a reload or restart to take effect.

```
[standalone@localhost:9990 /] cd core-service=management/access=authorization
[standalone@localhost:9990 access=authorization] :write-attribute(name=provider,value=rbac)
{
    "outcome" => "success",
    "response-headers" => {
        "operation-requires-reload" => true,
        "process-state" => "reload-required"
    }
}
[standalone@localhost:9990 access=authorization] reload
```

In a managed domain, the access control configuration is part of the domain wide configuration, so the resource address is the same as above, but the CLI is connected to the master Domain Controller:

```
[domain@localhost:9990 /] cd core-service=management/access=authorization
[domain@localhost:9990 access=authorization] :write-attribute(name=provider,value=rbac)
{
    "outcome" => "success",
    "response-headers" => {
        "operation-requires-reload" => true,
        "process-state" => "reload-required"
    },
    "result" => undefined,
    "server-groups" => {"main-server-group" => {"host" => {"master" => {
        "server-one" => {"response" => {
            "outcome" => "success",
            "response-headers" => {
                "operation-requires-reload" => true,
                "process-state" => "reload-required"
            }
        }},
        "server-two" => {"response" => {
            "outcome" => "success",
            "response-headers" => {
                "operation-requires-reload" => true,
                "process-state" => "reload-required"
            }
        }}
    }}}}
}
[domain@localhost:9990 access=authorization] reload --host=master
```

As with a standalone server, a reload or restart is required for the change to take effect. In this case, all hosts and servers in the domain will need to be reloaded or restarted, starting with the master Domain Controller, so be sure to plan well before making this change.

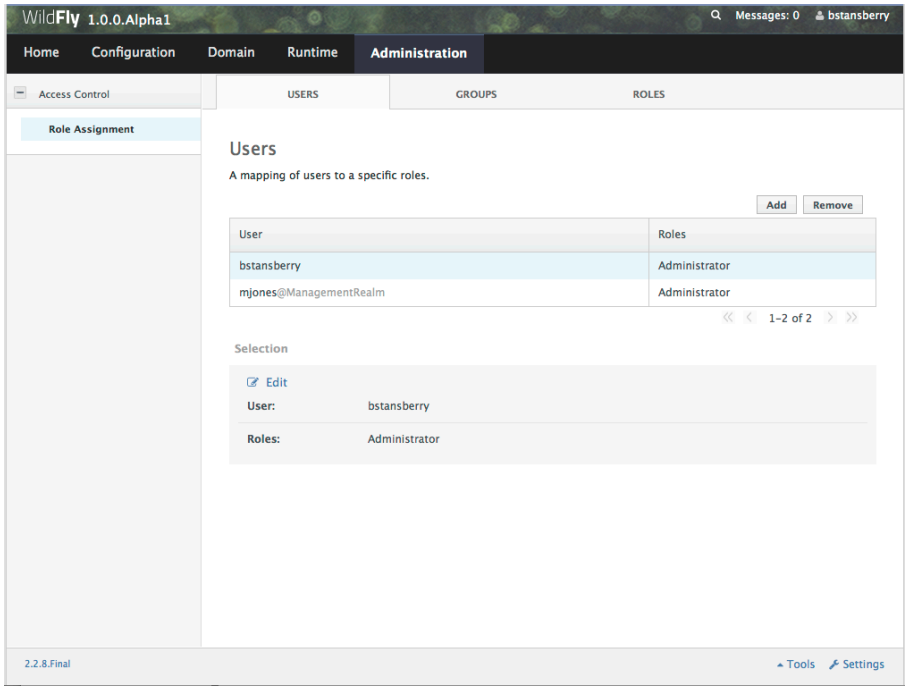
5.20.4 Mapping users and groups to roles

Once the "rbac" access control provider is enabled, only users who are mapped to one of the available roles will have any administrative permissions at all. So, to make RBAC useful, a mapping between individual users or groups of users and the available roles must be performed.

Mapping individual users

The easiest way to map individual users to roles is to use the web-based admin console.

Navigate to the "Administration" tab and the "Users" subtab. From there individual user mappings can be added, removed, or edited.



The CLI can also be used to map individuals users to roles.

First, if one does not exist, create the parent resource for all mappings for a role. Here we create the resource for the `Administrator` role.

```
[domain@localhost:9990 /]
/core-service=management/access=authorization/role-mapping=Administrator:add
{
    "outcome" => "success",
    "result" => undefined,
    "server-groups" => {"main-server-group" => {"host" => {"master" => {
        "server-one" => {"response" => {"outcome" => "success"}},
        "server-two" => {"response" => {"outcome" => "success"}}
    }}}}
}
```

Once this is done, map a user to the role:

```
[domain@localhost:9990 /]
/core-service=management/access=authorization/role-mapping=Administrator/include=user-jsmith:add(n
"outcome" => "success",
    "result" => undefined,
    "server-groups" => {"main-server-group" => {"host" => {"master" => {
        "server-one" => {"response" => {"outcome" => "success"}},
        "server-two" => {"response" => {"outcome" => "success"}}
    }}}}
}
```

Now if user `jsmith` authenticates to any security realm associated with the management interface they are using, he will be mapped to the `Administrator` role.

To restrict the mapping to a particular security realm, change the `realm` attribute to the realm name. This might be useful if different realms are associated with different management interfaces, and the goal is to limit a user to a particular interface.

```
[domain@localhost:9990 /]
/core-service=management/access=authorization/role-mapping=Administrator/include=user-mjones:add(n
"outcome" => "success",
    "result" => undefined,
    "server-groups" => {"main-server-group" => {"host" => {"master" => {
        "server-one" => {"response" => {"outcome" => "success"}},
        "server-two" => {"response" => {"outcome" => "success"}}
    }}}}
}
```

User groups

A "group" is an arbitrary collection of users that may exist in the end user environment. They can be named whatever the end user organization wants and can contain whatever users the end user organization wants. Some of the authentication store types supported by WildFly security realms include the ability to access information about what groups a user is a member of and associate this information with the `Subject` produced when the user is authenticated. This is currently supported for the following authentication store types:

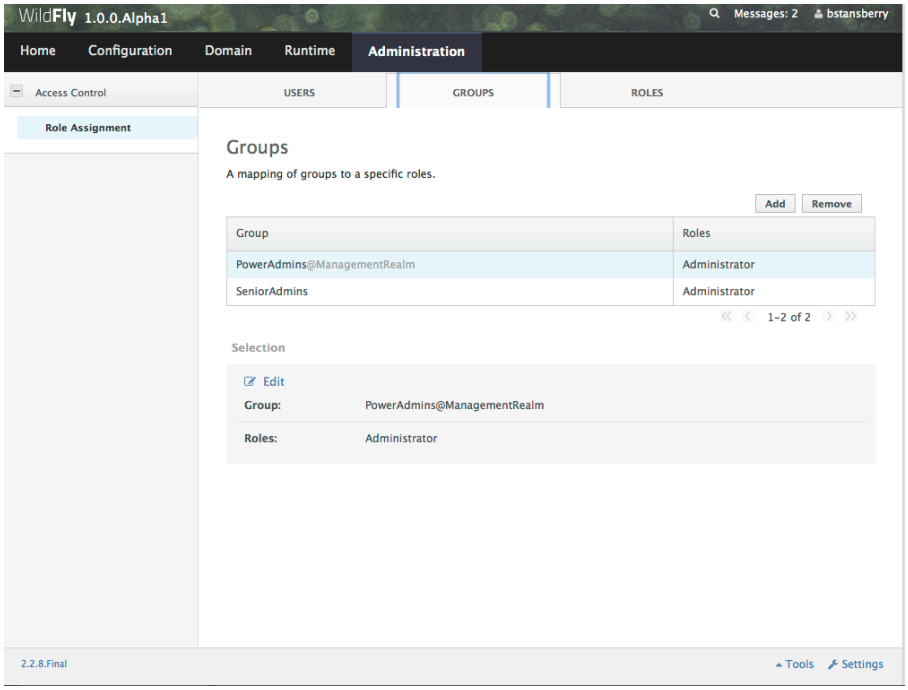
- properties file (via the `<realm_name>-groups.properties` file)
- LDAP (via directory-server-specific configuration)

Groups are convenient when it comes to associating a user with a role, since entire groups can be associated with a role in a single mapping.

Mapping groups to roles

The easiest way to map groups to roles is to use the web-based admin console.

Navigate to the "Administration" tab and the "Groups" subtab. From there group mappings can be added, removed, or edited.



The CLI can also be used to map groups to roles. The only difference to individual user mapping is the value of the `type` attribute should be `GROUP` instead of `USER`.

```
[domain@localhost:9990 /]
/core-service=management/access=authorization/role-mapping=Administrator/include=group-SeniorAdmin
"outcome" => "success",
    "result" => undefined,
    "server-groups" => {"main-server-group" => {"host" => {"master" => {
        "server-one" => {"response" => {"outcome" => "success"}},
        "server-two" => {"response" => {"outcome" => "success"}}
    }}}}
}
```

As with individual user mappings, the mapping can be restricted to users authenticating via a particular security realm:

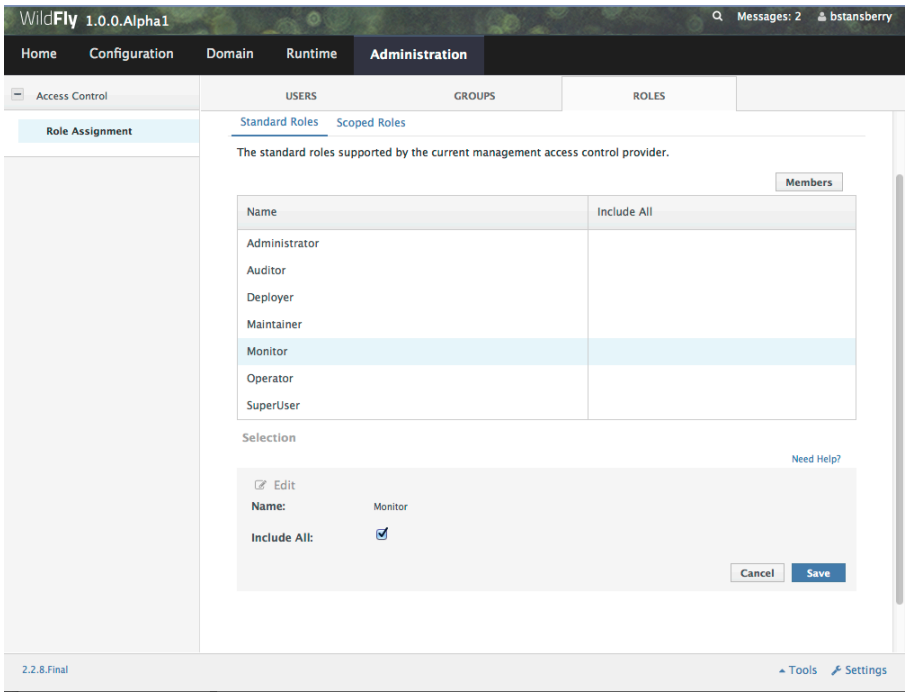
```
[domain@localhost:9990 /]
/core-service=management/access=authorization/role-mapping=Administrator/include=group-PowerAdmins
"outcome" => "success",
    "result" => undefined,
    "server-groups" => {"main-server-group" => {"host" => {"master" => {
        "server-one" => {"response" => {"outcome" => "success"}},
        "server-two" => {"response" => {"outcome" => "success"}}
    }}}}
}
```

Including all authenticated users in a role

It's possible to specify that all authenticated users should be mapped to a particular role. This could be used, for example, to ensure that anyone who can authenticate can at least have `Monitor` privileges.

> ⚠ A user who can authenticate to the management security realm but who does not map to a role will not be able to perform any administrative functions, not even reads.

In the web based admin console, navigate to the "Administration" tab, "Roles" subtab, highlight the relevant role, click the "Edit" button and click on the "Include All" checkbox:



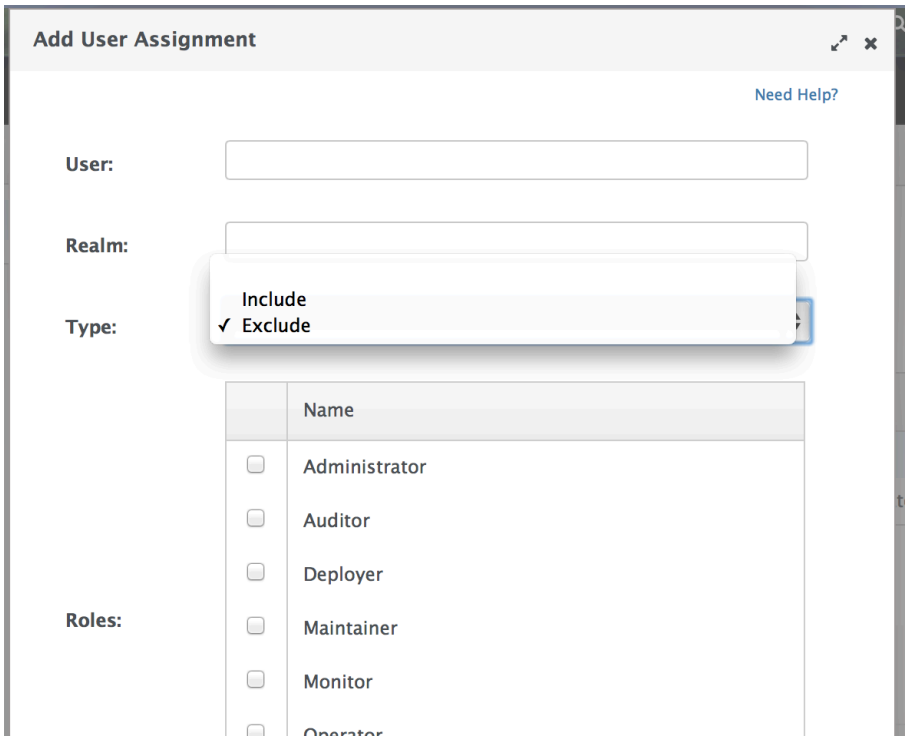
The same change can be made using the CLI:

```
[domain@localhost:9990 /]
/core-service=management/access=authorization/role-mapping=Monitor:write-attribute(name=include-al
"outcome" => "success",
    "result" => undefined,
    "server-groups" => {"main-server-group" => {"host" => {"master" => {
        "server-one" => {"response" => {"outcome" => "success"}},
        "server-two" => {"response" => {"outcome" => "success"}}
    }}}}
}
```

Excluding users and groups

It is also possible to explicitly exclude certain users and groups from a role. Exclusions take precedence over inclusions, including cases where the `include-all` attribute is set to true for a role.

In the admin console, excludes are done in the same screens as includes. In the add dialog, simply change the "Type" pulldown to "Exclude".



In the CLI, excludes are identical to includes, except the resource address has `exclude` instead of `include` as the key for the last address element:

```
[domain@localhost:9990 /]
/core-service=management/access=authorization/role-mapping=Monitor/exclude=group-Temps:add(name=Te
"outcome" => "success",
    "result" => undefined,
    "server-groups" => {"main-server-group" => {"host" => {"master" => {
        "server-one" => {"response" => {"outcome" => "success"}},
        "server-two" => {"response" => {"outcome" => "success"}}
    }}}}
}
```

Users who map to multiple roles

It is possible that a given user will be mapped to more than one role. When this occurs, by default the user will be granted the union of the permissions of the two roles. This behavior can be changed **on a global basis** to instead respond to the user request with an error if this situation is detected:

```
[standalone@localhost:9990 /] cd core-service=management/access=authorization
[standalone@localhost:9990 access=authorization]
:write-attribute(name=permission-combination-policy,value=rejecting)
{"outcome" => "success"}
```

Note that no reload is required; the change takes immediate effect.

To restore the default behavior, set the value to "permissive":

```
[standalone@localhost:9990 /] cd core-service=management/access=authorization
[standalone@localhost:9990 access=authorization]
:write-attribute(name=permission-combination-policy,value=permissive)
{"outcome" => "success"}
```

5.20.5 Adding custom roles in a managed domain

A managed domain may involve a variety of servers running different configurations and hosting different applications. In such an environment, it is likely that there will be different teams of administrators responsible for different parts of the domain. To allow organizations to grant permissions to only parts of a domain, WildFly's RBAC scheme allows for the creation of custom "scoped roles". Scoped roles are based on the seven standard roles, but with permissions limited to a portion of the domain – either to a set of server groups or to a set of hosts.

Server group scoped roles

The privileges for a server-group scoped role are constrained to resources associated with one or more server groups. Server groups are often associated with a particular application or set of applications; organizations that have separate teams responsible for different applications may find server-group scoped roles useful.

A server-group scoped role is equivalent to the default role upon which it is based, but with privileges constrained to target resources in the resource trees rooted in the server group resources. The server-group scoped role can be configured to include privileges for the following resources trees logically related to the server group:

- Profile
- Socket Binding Group
- Deployment
- Deployment override
- Server group
- Server config
- Server

Resources in the profile, socket binding group, server config and server portions of the tree that are not logically related to a server group associated with the server-group scoped role will not be addressable by a user in that role. So, in a domain with server groups “a” and “b”, a user in a server-group scoped role that grants access to “a” will not be able to address /server-group=b. The system will treat that resource as non-existent for that user.

In addition to these privileges, users in a server-group scoped role will have non-sensitive read privileges (equivalent to the Monitor role) for resources other than those listed above.

The easiest way to create a server-group scoped role is to use the admin console. But you can also use the CLI to create a server-group scoped role.

```
[domain@localhost:9990 /]
/core-service=management/access=authorization/server-group-scoped-role=MainGroupAdmins:add(base-ro
"outcome" => "success",
    "result" => undefined,
    "server-groups" => {"main-server-group" => {"host" => {"master" => {
        "server-one" => {"response" => {"outcome" => "success"}},
        "server-two" => {"response" => {"outcome" => "success"}}
    }}}}
}
```

Once the role is created, users or groups can be mapped to it the same as with the seven standard roles.

Host scoped roles

The privileges for a host-scoped role are constrained to resources associated with one or more hosts. A user with a host-scoped role cannot modify the domain wide configuration. Organizations may use host-scoped roles to give administrators relatively broad administrative rights for a host without granting such rights across the managed domain.

A host-scoped role is equivalent to the default role upon which it is based, but with privileges constrained to target resources in the resource trees rooted in the host resources for one or more specified hosts.

In addition to these privileges, users in a host-scoped role will have non-sensitive read privileges (equivalent to the Monitor role) for domain wide resources (i.e. those not in the /host=* section of the tree.)

Resources in the /host=* portion of the tree that are unrelated to the hosts specified for the Host Scoped Role will not be visible to users in that host-scoped role. So, in a domain with hosts “a” and “b”, a user in a host-scoped role that grants access to “a” will not be able to address /host=b. The system will treat that resource as non-existent for that user.

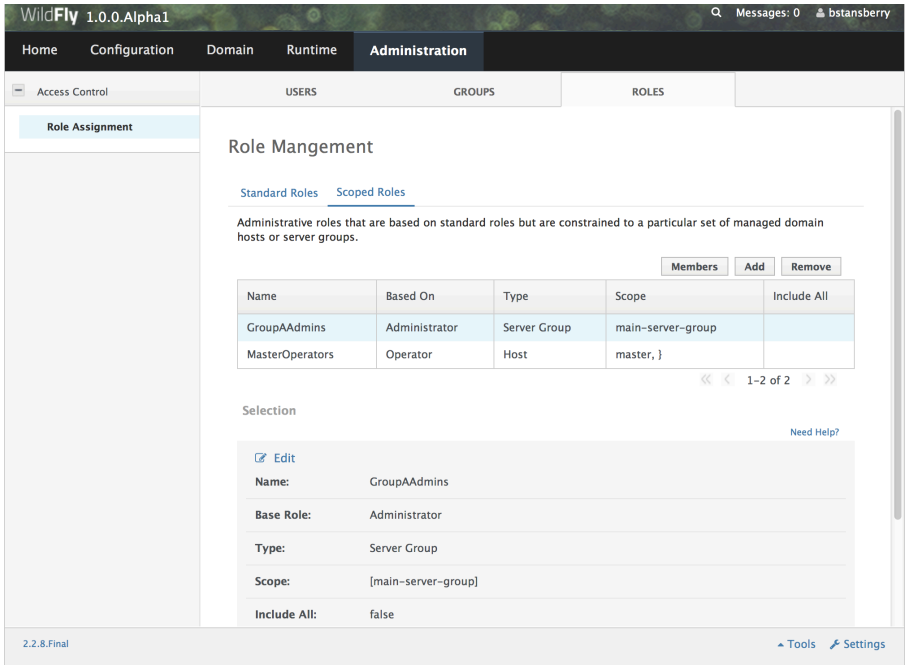
The easiest way to create a host-scoped role is to use the admin console. But you can also use the CLI to create a host scoped role.

```
[domain@localhost:9990 /]
/core-service=management/access=authorization/host-scoped-role=MasterOperators:add(base-role=Opera
"outcome" => "success",
    "result" => undefined,
    "server-groups" => {"main-server-group" => {"host" => {"master" => {
        "server-one" => {"response" => {"outcome" => "success"}},
        "server-two" => {"response" => {"outcome" => "success"}}
    }}}}
}
```

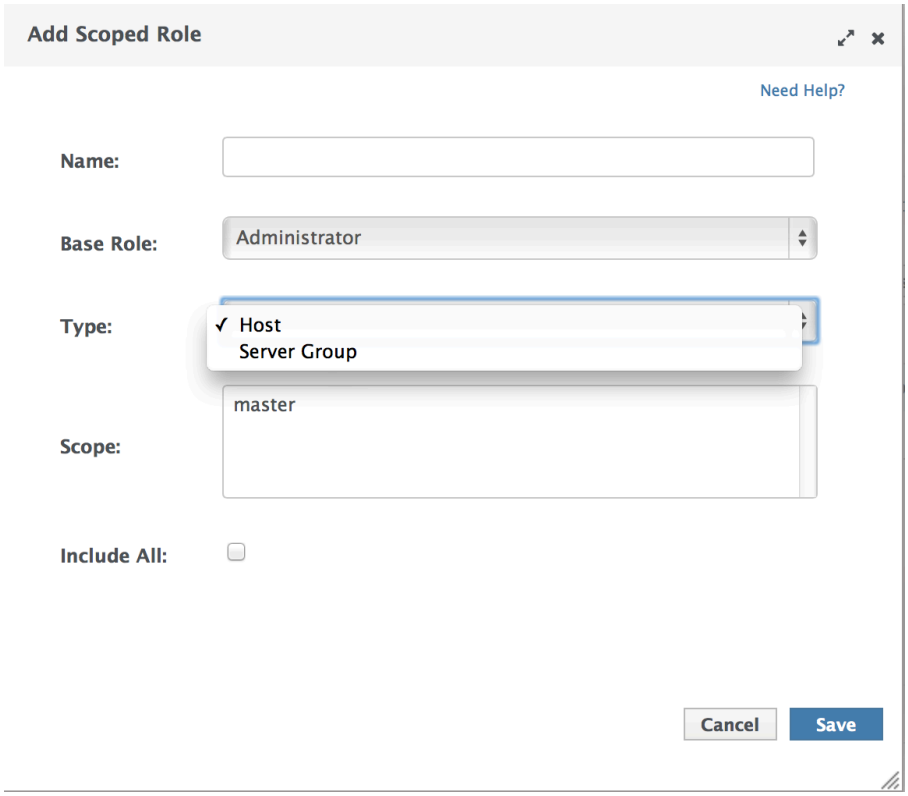
Once the role is created, users or groups can be mapped to it the same as with the seven standard roles.

Using the admin console to create scoped roles

Both server-group and host scoped roles can be added, removed or edited via the admin console. Select "Scoped Roles" from the "Administration" tab, "Roles" subtab:



When adding a new scoped role, use the dialogue's "Type" pull down to choose between a host scoped role and a server-group scoped role. Then place the names of the relevant hosts or server groups in the "Scope" text are.



5.20.6 Configuring constraints

The following factors are used to determine whether a given role is granted a permission:

- What the requested action is (address, read, write)
- Whether the resource, attribute or operation affects the persistent configuration
- Whether the resource, attribute or operation is related to the administrative audit logging function
- Whether the resource, attribute or operation is configured as security sensitive
- Whether an attribute or operation parameter value has a security vault expression
- Whether a resource is considered to be associated with applications, as opposed to being part of a general container configuration

The first three of these factors are non-configurable; the latter three allow some customization.

Configuring sensitivity

"Sensitivity" constraints are about restricting access to security-sensitive data. Different organizations may have different opinions about what is security sensitive, so WildFly provides configuration options to allow users to tailor these constraints.

Sensitive resources, attributes and operations

The developers of the WildFly core and of any subsystem may annotate resources, attributes or operations with a "sensitivity classification". Classifications are either provided by the core and may be applicable anywhere in the management model, or they are scoped to a particular subsystem. For each classification, there will be a setting declaring whether by default the addressing, read and write actions are considered to be sensitive. If an action is sensitive, only users in the roles able to deal with sensitive data (Administrator, Auditor, SuperUser) will have permissions.

Using the CLI, administrators can see the settings for a classification. For example, there is a core classification called "socket-config" that is applied to elements throughout the model that relate to configuring sockets:

```
[domain@localhost:9990 /] cd
core-service=management/access=authorization/constraint=sensitivity-classification/type=core/class
classification=socket-config] ls -l
ATTRIBUTE                      VALUE     TYPE
configured-requires-addressable undefined BOOLEAN
configured-requires-read        undefined BOOLEAN
configured-requires-write       undefined BOOLEAN
default-requires-addressable    false     BOOLEAN
default-requires-read           false     BOOLEAN
default-requires-write          true      BOOLEAN

CHILD      MIN-OCCURS MAX-OCCURS
applies-to n/a        n/a
```

The various `default-requires-...` attributes indicate whether a user must be in a role that allows security sensitive actions in order to perform the action. In the `socket-config` example above, `default-requires-write` is true, while the others are false. So, by default modifying a setting involving socket configuration is considered sensitive, while addressing those resources or doing reads is not sensitive.

The `default-requires-...` attributes are read-only. The `configured-requires-...` attributes however can be modified to override the default settings with ones appropriate for your organization. For example, if your organization doesn't regard modifying socket configuration settings to be security sensitive, you can change that setting:

```
[domain@localhost:9990 classification=socket-config]
:write-attribute(name=configured-requires-write,value=false)
{
    "outcome" => "success",
    "result" => undefined,
    "server-groups" => {"main-server-group" => {"host" => {"master" => {
        "server-one" => {"response" => {"outcome" => "success"}},
        "server-two" => {"response" => {"outcome" => "success"}}
    }}}}
}
```

Administrators can also read the management model to see to which resources, attributes and operations a particular sensitivity classification applies:

```
[domain@localhost:9990 classification=socket-config]
:read-children-resources(child-type=applies-to)
{
    "outcome" => "success",
    "result" => {
        "/host=master" => {
            "address" => "/host=master",
            "attributes" => [],
            "entire-resource" => false,
            "operations" => ["resolve-internet-address"]
        },
        "/host=master/core-service=host-environment" => {
            "address" => "/host=master/core-service=host-environment",
            "attributes" => [
                "host-controller-port",
                "host-controller-address",
                "process-controller-port",
                "process-controller-address"
            ],
            "entire-resource" => false,
            "operations" => []
        },
        "/host=master/core-service=management/management-interface=http-interface" => {
            "address" =>
"/host=master/core-service=management/management-interface=http-interface",
            "attributes" => [
                "port",
                "secure-interface",
```

```
                "secure-port",
                "interface"
            ],
            "entire-resource" => false,
            "operations" => []
        },
        "/host=master/core-service=management/management-interface=native-interface" => {
            "address" =>
"/host=master/core-service=management/management-interface=native-interface",
            "attributes" => [
                "port",
                "interface"
            ],
            "entire-resource" => false,
            "operations" => []
        },
        "/host=master/interface=*" => {
            "address" => "/host=master/interface=*",
            "attributes" => [],
            "entire-resource" => true,
            "operations" => ["resolve-internet-address"]
        },
        "/host=master/server-config=*/interface=*" => {
            "address" => "/host=master/server-config=*/interface=*",
            "attributes" => [],
            "entire-resource" => true,
            "operations" => []
        },
        "/interface=*" => {
            "address" => "/interface=*",
            "attributes" => [],
            "entire-resource" => true,
            "operations" => []
        },
        "/profile=*/subsystem=messaging/hornetq-server=*/broadcast-group=*" => {
            "address" => "/profile=*/subsystem=messaging/hornetq-server=*/broadcast-group=*",
            "attributes" => [
                "group-address",
                "group-port",
                "local-bind-address",
                "local-bind-port"
            ],
            "entire-resource" => false,
            "operations" => []
        },
        "/profile=*/subsystem=messaging/hornetq-server=*/discovery-group=*" => {
            "address" => "/profile=*/subsystem=messaging/hornetq-server=*/discovery-group=*",
            "attributes" => [
                "group-address",
                "group-port",
                "local-bind-address"
            ],
            "entire-resource" => false,
            "operations" => []
        },
        "/profile=*/subsystem=transactions" => {
            "address" => "/profile=*/subsystem=transactions",
            "attributes" => ["process-id-socket-max-ports"],
```

```
            "entire-resource" => false,
            "operations" => []
        },
        "/server-group=*" => {
            "address" => "/server-group=*",
            "attributes" => ["socket-binding-port-offset"],
            "entire-resource" => false,
            "operations" => []
        },
        "/socket-binding-group=*" => {
            "address" => "/socket-binding-group=*",
            "attributes" => [],
            "entire-resource" => true,
            "operations" => []
        }
    }
}
```

There will be a separate child for each address to which the classification applies. The `entire-resource` attribute will be true if the classification applies to the entire resource. Otherwise, the `attributes` and `operations` attributes will include the names of attributes or operations to which the classification applies.

Classifications with broad use

Several of the core sensitivity classifications are commonly used across the management model and deserve special mention.

Name	Description
credential	An attribute whose value is some sort of credential, e.g. a password or a username. By default sensitive for both reads and writes
security-domain-ref	An attribute whose value is the name of a security domain. By default sensitive for both reads and writes
security-realm-ref	An attribute whose value is the name of a security realm. By default sensitive for both reads and writes
socket-binding-ref	An attribute whose value is the name of a socket binding. By default not sensitive for any action
socket-config	A resource, attribute or operation that somehow relates to configuring a socket. By default sensitive for writes

Values with security vault expressions

By default any attribute or operation parameter whose value includes a security vault expression will be treated as sensitive, even if no sensitivity classification applies or the classification does not treat the action as sensitive.

This setting can be **globally** changed via the CLI. There is a resource for this configuration:

```
[domain@localhost:9990 /] cd
core-service=management/access=authorization/constraint=vault-expression
[domain@localhost:9990 constraint=vault-expression] ls -l
ATTRIBUTE                 VALUE     TYPE
configured-requires-read  undefined BOOLEAN
configured-requires-write undefined BOOLEAN
default-requires-read     true      BOOLEAN
default-requires-write    true      BOOLEAN
```

The various `default-requires-...` attributes indicate whether a user must be in a role that allows security sensitive actions in order to perform the action. So, by default both reading and writing attributes whose values include vault expressions requires a user to be in one of the roles with sensitive data permissions.

The `default-requires-...` attributes are read-only. The `configured-requires-...` attributes however can be modified to override the default settings with settings appropriate for your organization. For example, if your organization doesn't regard reading vault expressions to be security sensitive, you can change that setting:

```
[domain@localhost:9990 constraint=vault-expression]
:write-attribute(name=configured-requires-read,value=false)
{
    "outcome" => "success",
    "result" => undefined,
    "server-groups" => {"main-server-group" => {"host" => {"master" => {
        "server-one" => {"response" => {"outcome" => "success"}},
        "server-two" => {"response" => {"outcome" => "success"}}
    }}}}
}
```

This vault-expression constraint overlaps somewhat with the core "credential" sensitivity classification in that the most typical uses of a vault expression are in attributes that contain a user name or password, and those will typically be annotated with the "credential" sensitivity classification. So, if you change the settings for the "credential" sensitivity classification you may also need to make a corresponding change to the vault-expression constraint settings, or your change will not have full effect.

Be aware though, that vault expressions can be used in any attribute that supports expressions, not just in credential-type attributes. So it is important to be familiar with where and how your organization uses vault expressions before changing these settings.

Configuring "Deployer" role access

The standard Deployer role has its write permissions limited to resources that are considered to be "application resources"; i.e. conceptually part of an application and not part of the general server configuration. By default, only deployment resources are considered to be application resources. However, different organizations may have different opinions on what qualifies as an application resource, so for resource types that subsystems authors consider *potentially* to be application resources, WildFly provides a configuration option to declare them as such. Such resources will be annotated with an "application classification".

For example, the mail subsystem provides such a classification:

```
[domain@localhost:9990 /] cd
/core-service=management/access=authorization/constraint=application-classification/type=mail/clas
classification=mail-session] ls -l
ATTRIBUTE              VALUE     TYPE
configured-application undefined BOOLEAN
default-application    false     BOOLEAN

CHILD      MIN-OCCURS MAX-OCCURS
applies-to n/a        n/a
```

Use `read-resource` or `read-children-resources` to see what resources have this classification applied:

```
[domain@localhost:9990 classification=mail-session]
:read-children-resources(child-type=applies-to)
{
    "outcome" => "success",
    "result" => {"/profile=*/subsystem=mail/mail-session=*" => {
        "address" => "/profile=*/subsystem=mail/mail-session=*",
        "attributes" => [],
        "entire-resource" => true,
        "operations" => []
    }}
}
```

This indicates that this classification, intuitively enough, only applies to mail subsystem mail-session resources.

To make resources with this classification writeable by users in the Deployer role, set the `configured-application` attribute to true.

```
[domain@localhost:9990 classification=mail-session]
:write-attribute(name=configured-application,value=true)
{
    "outcome" => "success",
    "result" => undefined,
    "server-groups" => {"main-server-group" => {"host" => {"master" => {
        "server-one" => {"response" => {"outcome" => "success"}},
        "server-two" => {"response" => {"outcome" => "success"}}
    }}}}
}
```

Application classifications shipped with WildFly

The subsystems shipped with the full WildFly distribution include the following application classifications:

Subsystem	Classification
datasources	data-source
datasources	jdbc-driver
datasources	xa-data-source
logging	logger
logging	logging-profile
mail	mail-session
messaging	jms-queue
messaging	jms-topic
messaging	queue
messaging	security-setting
naming	binding
resource-adapters	resource-adapter
security	security-domain

In each case the classification applies to the resources you would expect, given its name.

5.20.7 RBAC effect on administrator user experience

The RBAC scheme will result in reduced permissions for administrators who do not map to the SuperUser role, so this will of course have some impact on their experience when using administrative tools like the admin console and the CLI.

Admin console

The admin console takes great pains to provide a good user experience even when the user has reduced permissions. Resources the user is not permitted to see will simply not be shown, or if appropriate will be replaced in the UI with an indication that the user is not authorized. Interaction units like "Add" and "Remove" buttons and "Edit" links will be suppressed if the user has no write permissions.

CLI

The CLI is a much more unconstrained tool than the admin console is, allowing users to try to execute whatever operations they wish, so it's more likely that users who attempt to do things for which they lack necessary permissions will receive failure messages. For example, a user in the Monitor role cannot read passwords:

```
[domain@localhost:9990 /]
/profile=default/subsystem=datasources/data-source=ExampleDS:read-attribute(name=password)
{
    "outcome" => "failed",
    "result" => undefined,
    "failure-description" => "WFLYCTL0313: Unauthorized to execute operation 'read-attribute'
for resource '[
    (\"profile\" => \"default\"),
    (\"subsystem\" => \"datasources\"),
    (\"data-source\" => \"ExampleDS\")
]' -- \"WFLYCTL0332: Permission denied\"",
    "rolled-back" => true
}
```

If the user isn't even allowed to address the resource then the response would be as if the resource doesn't exist, even though it actually does:

```
[domain@localhost:9990 /]
/profile=default/subsystem=security/security-domain=other:read-resource
{
    "outcome" => "failed",
    "failure-description" => "WFLYCTL0216: Management resource '[
    (\"profile\" => \"default\"),
    (\"subsystem\" => \"security\"),
    (\"security-domain\" => \"other\")
]' not found",
    "rolled-back" => true
}
```

This prevents unauthorized users fishing for sensitive data in resource addresses by checking for "Permission denied" type failures.

Users who use the `read-resource` operation may ask for data, some of which they are allowed to see and some of which they are not. If this happens, the request will not fail, but inaccessible data will be elided and a response header will be included advising on what was not included. Here we show the effect of a Monitor trying to recursively read the security subsystem configuration:

```
[domain@localhost:9990 /] /profile=default/subsystem=security:read-resource(recursive=true)
{
    "outcome" => "success",
    "result" => {
        "deep-copy-subject-mode" => undefined,
        "security-domain" => undefined,
        "vault" => undefined
    },
    "response-headers" => {"access-control" => [{
        "absolute-address" => [
            ("profile" => "default"),
            ("subsystem" => "security")
        ],
        "relative-address" => [],
        "filtered-attributes" => ["deep-copy-subject-mode"],
        "filtered-children-types" => ["security-domain"]
    }]}
}
```

The `response-headers` section includes access control data in a list with one element per relevant resource. (In this case there's just one.) The absolute and relative address of the resource is shown, along with the fact that the value of the `deep-copy-subject-mode` attribute has been filtered (i.e. undefined is shown as the value, which may not be the real value) as well as the fact that child resources of type `security-domain` have been filtered.

Description of access control constraints in the management model metadata

The management model descriptive metadata returned from operations like `read-resource-description` and `read-operation-description` can be configured to include information describing the access control constraints relevant to the resource, This is done by using the `access-control` parameter. The output will be tailored to the caller's permissions. For example, a user who maps to the Monitor role could ask for information about a resource in the mail subsystem:

```
[domain@localhost:9990 /] cd /profile=default/subsystem=mail/mail-session=default/server=smtp
[domain@localhost:9990 server=smtp] :read-resource-description(access-control=trim-descriptions)
{
    "outcome" => "success",
    "result" => {
        "description" => undefined,
        "access-constraints" => {"application" => {"mail-session" => {"type" => "mail"}}},
        "attributes" => undefined,
        "operations" => undefined,
        "children" => {},
        "access-control" => {
            "default" => {
                "read" => true,
                "write" => false,
                "attributes" => {
                    "outbound-socket-binding-ref" => {
                        "read" => true,
                        "write" => false
                    },
                    "username" => {
                        "read" => false,
                        "write" => false
                    },
                    "tls" => {
                        "read" => true,
                        "write" => false
                    },
                    "ssl" => {
                        "read" => true,
                        "write" => false
                    },
                    "password" => {
                        "read" => false,
                        "write" => false
                    }
                }
            },
            "exceptions" => {}
        }
    }
}
```

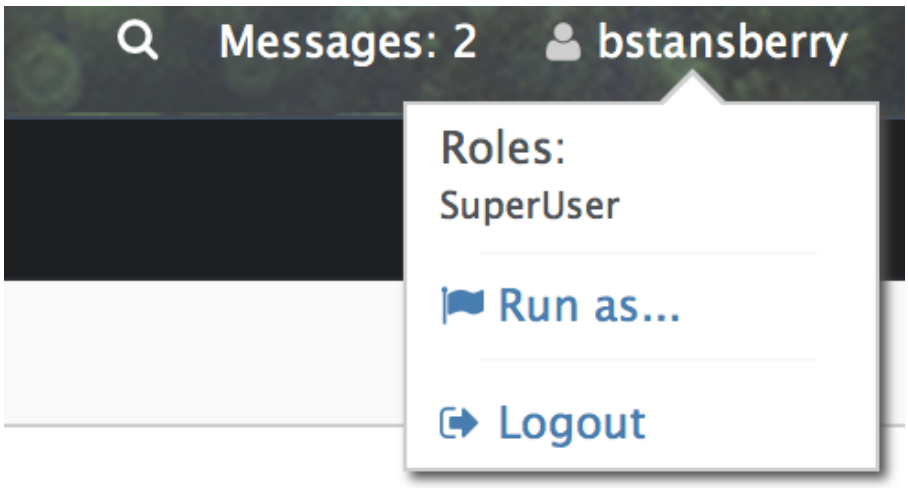
Because `trim-descriptions` was used as the value for the `access-control` parameter, the typical "description", "attributes", "operations" and "children" data is largely suppressed. (For more on this, see below.) The `access-constraints` field indicates that this resource is annotated with an [application constraint]. The `access-control` field includes information about the permissions the current caller has for this resource. The `default` section shows the default settings for resources of this type. The `read` and `write` fields directly under `default` show that the caller can, in general, read this resource but cannot write it. The `attributes` section shows the individual attribute settings. Note that Monitor cannot read the `username` and `password` attributes.

There are three valid values for the `access-control` parameter to `read-resource-description` and `read-operation-description`:

- **none** – do not include access control information in the response. This is the default behavior if no parameter is included.
- **trim-descriptions** – remove the normal description details, as shown in the example above
- **combined-descriptions** – include both the normal output and the access control data

5.20.8 Learning about your own role mappings

Users can learn in which roles they are operating. In the admin console, click on your name in the top right corner; the roles you are in will be shown.



CLI users should use the `whoami` operation with the `verbose` attribute set:

```
[domain@localhost:9990 /] :whoami(verbose=true)
{
    "outcome" => "success",
    "result" => {
        "identity" => {
            "username" => "aadams",
            "realm" => "ManagementRealm"
        },
        "mapped-roles" => [
            "Maintainer"
        ]
    }
}
```

5.20.9 "Run-as" capability for SuperUsers

If a user maps to the SuperUser role, WildFly also supports letting that user request that they instead map to one or more other roles. This can be useful when doing demos, or when the SuperUser is changing the RBAC configuration and wants to see what effect the changes have from the perspective of a user in another role. This capability is only available to the SuperUser role, so it can only be used to narrow a user's permissions, not to potentially increase them.

CLI run-as

With the CLI, run-as capability is on a per-request basis. It is done by using the "roles" operation header, the value of which can be the name of a single role or a bracket-enclosed, comma-delimited list of role names.

Example with a low level operation:

```
[standalone@localhost:9990 /] :whoami(verbose=true){roles=[Operator,Auditor]}
{
    "outcome" => "success",
    "result" => {
        "identity" => {
            "username" => "$local",
            "realm" => "ManagementRealm"
        },
        "mapped-roles" => [
            "Auditor",
            "Operator"
        ]
    }
}
```

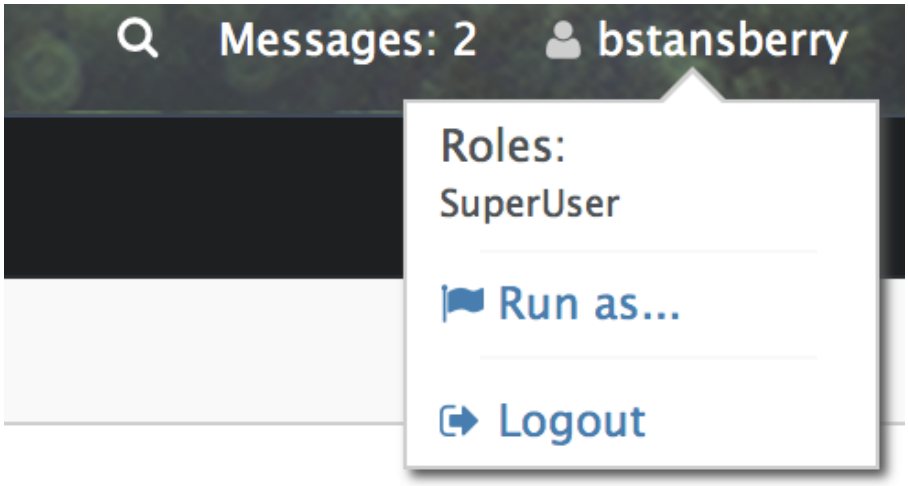
Example with a CLI command:

```
[standalone@localhost:9990 /] deploy /tmp/helloworld.war --headers={roles=Monitor}
{"WFLYCTL0062: Composite operation failed and was rolled back. Steps that failed:" =>
{"Operation step-1" => "WFLYCTL0313: Unauthorized to execute operation 'add' for resource
'[(\"deployment\" => \"helloworld.war\")]' -- \"WFLYCTL0332: Permission denied\""}}
[standalone@localhost:9990 /] deploy /tmp/helloworld.war --headers={roles=Maintainer}
```

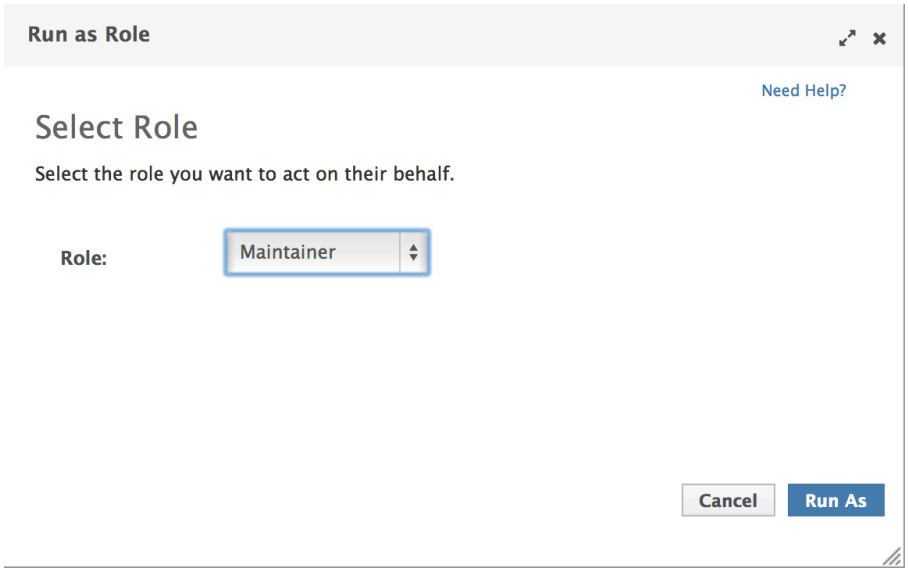
Here we show the effect of switching to a role that isn't granted the necessary permission.

Admin console run-as

Admin console users can change the role in which they operate by clicking on their name in the top right corner and clicking on the "Run as..." link.



Then select the role in which you wish to operate:



The console will need to be restarted in order for the change to take effect.

Using run-as roles with the "simple" access control provider

This "run-as" capability is available even if the "simple" access control provider is used. When the "simple" provider is used, any authenticated administrator is treated the same as if they would map to SuperUser when the "rbac" provider is used.
However, the "simple" provider actually understands all of the "rbac" provider configuration settings described above, but only makes use of them if the "run-as" capability is used for a request. Otherwise, the SuperUser role has all permissions, so detailed configuration is irrelevant.

Using the run-as capability with the "simple" provider may be useful if an administrator is setting up an rbac provider configuration before switching the provider to rbac to make that configuration take effect. The administrator can then run-as different roles to see the effect of the planned settings.

5.21 Security Realms

Within WildFly we make use of security realms to secure access to the management interfaces, these same realms are used to secure inbound access as exposed by JBoss Remoting such as remote JNDI and EJB access, the realms are also used to define an identity for the server - this identity can be used for both inbound connections to the server and outbound connections being established by the server.

5.21.1 General Structure

The general structure of a management realm definition is: -

```
<security-realm name="ManagementRealm">
   <plug-ins></plug-ins>
   <server-identities></server-identities>
   <authentication></authentication>
   <authorization></authorization>
</security-realm>
```

- `plug-ins` - This is an optional element that is used to define modules what will be searched for security realm PlugInProviders to extend the capabilities of the security realms.
- `server-identities` - An optional element to define the identity of the server as visible to the outside world, this applies to both inbound connection to a resource secured by the realm and to outbound connections also associated with the realm.

One example is the SSL identity of the server, for inbound connections this will control the identity of the server as the SSL connection is established, for outbound connections this same identity can be used where CLIENT-CERT style authentication is being performed.

A second example is where the server is establishing an outbound connection that requires username / password authentication - this element can be used to define that password.

- `authentication` - This is probably the most important element that will be used within a security realm definition and mostly applies to inbound connections to the server, this element defines which backing stores will be used to provide the verification of the inbound connection.

This element is optional as there are some scenarios where it will not be required such as if a realm is being defined for an outbound connection using a username and password.

- `authorization` - This is the final optional element and is used to define how roles are loaded for an authenticated identity. At the moment this is more applicable for realms used for access to EE deployments such as web applications or EJBs but this will also become relevant as we add role based authorization checks to the management model.

5.21.2 Using a Realm

After a realm has been defined it needs to be associated with an inbound or outbound connection for it to be used, the following are some examples where these associations are used within the WildFly 8 configuration.

Inbound Connections

Management Interfaces

Either within the `standalone.xml` or `host.xml` configurations the security realms can be associated with the management interface as follows:

```
<http-interface security-realm="ManagementRealm">...</http-interface>
```

If the `security-realm` attribute is omitted or removed from the interface definition it means that access to that interface will be unsecured.

By default we do bind these interfaces to the loopback address so that the interfaces are not accessible remotely out of the box, however do be aware that if these interfaces are then unsecured any other local user will be able to control and administer the WildFly installation.

Remoting Subsystem

The Remoting subsystem exposes a connector to allow for inbound comunications with JNDI and the EJB subsystem by default we associate the `ApplicationRealm` with this connection.

```
<subsystem xmlns="urn:jboss:domain:remoting:3.0">
  <endpoint worker="default"/>
  <http-connector name="http-remoting-connector" connector-ref="default"
security-realm="ApplicationRealm"/>
</subsystem>
```

Outbound Connections

Remoting Subsystem

Outbound connections can also be defined within the Remoting subsystem, these are typically used for remote EJB invocations from one AS server to another, in this scenario the security realm is used to obtain the server identity either it's password for X.509 certificate and possibly a trust store to verify the certificate of the remote host.

Even if the referenced realm contains username and password authentication configuration the client side of the connection will NOT use this to verify the remote server.

```
<remote-outbound-connection name="remote-ejb-connection"
                            outbound-socket-binding-ref="binding-remote-ejb-connection"
                            username="user1"
                            security-realm="PasswordRealm">
```

The security realm is only used to obtain the password for this example, as you can see here the username is specified separately.

Slave Host Controller

When running in domain mode slave host controllers need to establish a connection to the native interface of the master domain controller so these also need a realm for the identity of the slave.

```
<domain-controller>
  <remote host="${jboss.domain.master.address}" port="${jboss.domain.master.port:9999}"
security-realm="ManagementRealm"/>
</domain-controller>
```

By default when a slave host controller authenticates against the master domain controller it uses its configured name as its username. If you want to override the username used for authentication a `username` attribute can be added to the `<remote />` element.

5.21.3 Authentication

One of the primary functions of the security realms is to define the user stores that will be used to verify the identity of inbound connections, the actual approach taken at the transport level is based on the capabilities of these backing store definitions. The security realms are used to secure inbound connections for both the http management interface and for inbound remoting connections for both the native management interface and to access other services exposed over remoting - because of this there are some small differences between how the realm is used for each of these.

At the transport level we support the following authentication mechanisms.

HTTP	Remoting (SASL)
None	Anonymous
N/A	JBoss Local User
Digest	Digest
Basic	Plain
Client Cert	Client Cert

The most notable are the first two in this list as they need some additional explanation - the final 3 are fairly standard mechanisms.

If either the http interface, the native interface or a remoting connection are difined **without** a security realm reference then they are effectively unsecured, in the case of the http interface this means that no authentication will be performed on the incoming connection - for the remoting connections however we make use of SASL so require at least one authentication mechanism so make use of the anonymous mechanism to allow a user in without requiring a validated authentication process.

The next mechanism 'JBoss Local User' is specific to the remoting connections - as we ship WildFly secured by default we wanted a way to allow users to connect to their own AS installation after it is started without mandating that they define a user with a password - to accomplish this we have added the 'JBoss Local User' mechanism. This mechanism makes the use of tokens exchanged on the filesystem to prove that the client is local to the AS installation and has the appropriate file permissions to read a token written by the AS to file. As this mechanism is dependent on both server and client implementation details it is only supported for the remoting connections and not the http connections - at some point we may review if we can add support for this to the http interface but we would need to explore the options available with the commony used web browsers that are used to communicate with the http interface.

The Digest mechanism is simply the HTTP Digest / SASL Digest mechanism that authenticates the user by making use of md5 hashed including nonces to avoid sending passwords in plain text over the network - this is the preferred mechanism for username / password authentication.

The HTTP Basic / SASL Plain mechanism is made available for times that Digest can not be used but effectively this means that the users password will be sent over the network in the clear unless SSL is enabled.

The final mechanism Client-Cert allows X.509 certificates to be used to verify the identity of the remote client.

One point bearing in mind is that it is possible that an association with a realm can mean that a single incoming connection has the ability to choose between one or more authentication mechanisms. As an example it is possible that an incoming remoting connection could choose between 'Client Cert', A username password mechanism or 'JBoss Local User' for authentication - this would allow say a local user to use the local mechanism, a remote user to supply their username and password whilst a remote script could make a call and authenticate using a certificate.

5.21.4 Authorization

The actual security realms are not involved in any authorization decisions. However, they can be configured to load a user's roles, which will subsequently be used to make authorization decisions - when references to authorization are seen in the context of security realms, it is this loading of roles that is being referred to.

For the loading of roles, the process is split out to occur after the authentication step so after a user has been authenticated, a second step will occur to load the roles based on the username they used to authenticate with.

5.21.5 Out Of The Box Configuration

Before describing the complete set of configuration options available within the realms, we will look at the default configuration, as for most users, that is going to be the starting point before customising further.

The examples here are taken from the standalone configuration. However, the descriptions are equally applicable to domain mode. One point worth noting is that all security realms defined in the `host.xml` are available to be referenced within the domain configuration for the servers running on that host controller.

Management Realm

```
<security-realm name="ManagementRealm">
  <authentication>
    <local default-user="$local"/>
    <properties path="mgmt-users.properties" relative-to="jboss.server.config.dir"/>
  </authentication>
</security-realm>
```

The realm `ManagementRealm` is the simplest realm within the default configuration. This realm simply enables two authentication mechanisms, the local mechanism and username/password authentication which will be using Digest authentication.

- local

When using the local mechanism, it is optional for remote clients to send a username to the server. This configuration specifies that where clients do not send a username, it will be assumed that the clients username is `$local` - the `<local />` element can also be configured to allow other usernames to be specified by remote clients. However, for the default configuration, this is not enabled so is not supported.

- properties

For username / password authentication the users details will be loaded from the file `mgmt-users.properties` which is located in `{jboss.home}/standalone/configuration` or `{jboss.home}/domain/configuration` depending on the running mode of the server.

Each user is represented on their own line and the format of each line is `username=HASH` where `HASH` is a pre-prepared hash of the users password along with their username and the name of the realm which in this case is `ManagementRealm`.

You do not need to worry about generating the entries within the properties file as we provide a utility `add-user.sh` or `add-user.bat` to add the users, this utility is described in more detail below.

By pre-hashing the passwords in the properties file it does mean that if the user has used the same password on different realms then the contents of the file falling into the wrong hands does not nescesarily mean all accounts are compromised. **HOWEVER** the contents of the files do still need to be protected as they can be used to access any server where the realm name is the same and the user has the same username and password pair.

Application Realm

```
<security-realm name="ApplicationRealm">
  <authentication>
    <local default-user="$local" allowed-users="*"/>
    <properties path="application-users.properties" relative-to="jboss.server.config.dir"/>
  </authentication>
  <authorization>
    <properties path="application-roles.properties" relative-to="jboss.server.config.dir"/>
  </authorization>
</security-realm>
```

The realm `ApplicationRealm` is a slightly more complex realm as this is used for both

Authentication

The authentication configuration is very similar to the `ManagementRealm` in that it enabled both the local mechanism and a username/password based Digest mechanism.

- local

The local configuration is similar to the `ManagementRealm` in that where the remote user does not supply a username it will be assumed that the username is `$local`, however in addition to this there is now an `allowed-users` attribute with a value of `'*'` - this means that the remote user can specify any username and it will be accepted over the local mechanism provided that the local verification is a success.

To restrict the usernames that can be specified by the remote user a comma separated list of usernames can be specified instead within the `allowed-users` attribute.

- properties

The properties definition works in exactly the same way as the definition for `ManagementRealm` except now the properties file is called `application-users.properties`.

Authorization

The contents of the `Authorization` element are specific to the `ApplicationRealm`, in this case a properties file is used to load a users roles.

The properties file is called `application-roles.properties` and is located in `{jboss.home}/standalone/configuration` or `{jboss.home}/domain/configuration` depending on the running mode of the server. The format of this file is `username=ROLES` where *ROLES* is a comma separated list of the users roles.

As the loading of a users roles is a second step this is where it may be desirable to restrict which users can use the local mechanism so that some users still require username and password authentication for their roles to be loaded.

other security domain

```
<security-domain name="other" cache-type="default">
  <authentication>
    <login-module code="Remoting" flag="optional">
      <module-option name="password-stacking" value="useFirstPass"/>
    </login-module>
    <login-module code="RealmDirect" flag="required">
      <module-option name="password-stacking" value="useFirstPass"/>
    </login-module>
  </authentication>
</security-domain>
```

When applications are deployed to the application server they are associated with a security domain within the security subsystem, the `other` security domain is provided to work with the `ApplicationRealm`, this domain is defined with a pair of login modules Remoting and RealmDirect.

- Remoting

The `Remoting` login module is used to check if the request currently being authenticated is a request received over a Remoting connection, if so the identity that was created during the authentication process is used and associated with the current request.

If the request did not arrive over a Remoting connection this module does nothing and allows the JAAS based login to continue to the next module.

- RealmDirect

The `RealmDirect` login module makes use of a security realm to authenticate the current request if that did not occur in the `Remoting` login module and then use the realm to load the users roles, by default this login module assumes the realm to use is called `ApplicationRealm` although other names can be overridden using the "realm" module-option.

The advantage of this approach is that all of the backing store configuration can be left within the realm with the security domain just delegating to the realm.

5.21.6 user.sh

For use with the default configuration we supply a utility `add-user` which can be used to manage the properties files for the default realms used to store the users and their roles.

The add-user utility can be used to manage both the users in the `ManagementRealm` and the users in the `ApplicationRealm`, changes made apply to the properties file used both for domain mode and standalone mode.

After you have installed your application server and decided if you are going to run in standalone mode or domain mode you can delete the parent folder for the mode you are not using, the add-user utility will then only be managing the properties file for the mode in use.

The add-user utility is a command line utility however it can be run in both interactive and non-interactive mode. Depending on your platform the script to run the add-user utility is either `add-user.sh` or `add-user.bat` which can be found in `{jboss.home}/bin`.

This guide now contains a couple of examples of this utility in use to accomplish the most common tasks.

Adding a User

Adding users to the properties files is the primary purpose of this utility. Usernames can only contain the following characters in any number and in any order:

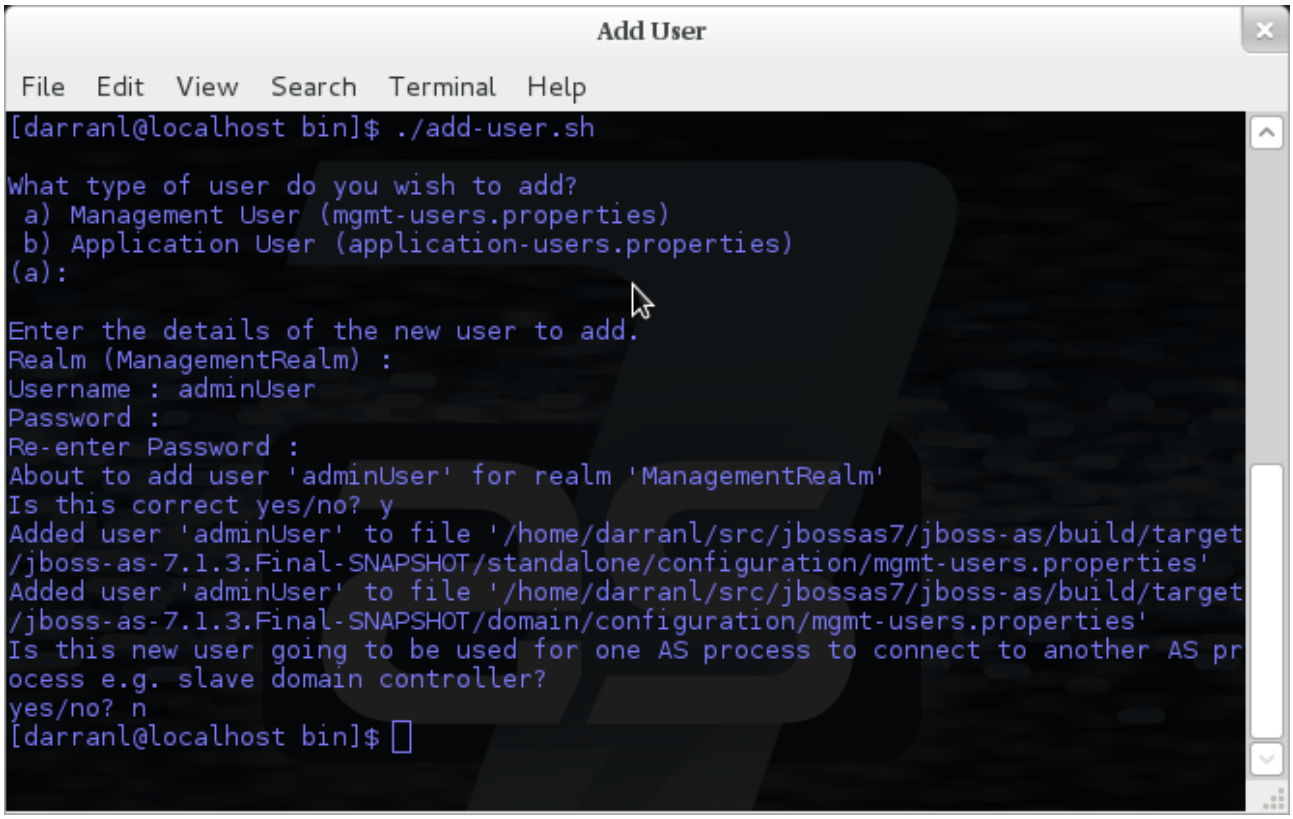
- Alphanumeric characters (a-z, A-Z, 0-9)
- Dashes (-), periods (.), commas (,), at (@)
- Escaped backslash (\\ )
- Escaped equals (\=)

The server caches the contents of the properties files in memory, however the server does check the modified time of the properties files on each authentication request and re-load if the time has been updated - this means all changes made by this utility are immediately applied to any running server.

A Management User

The default name of the realm for management users is `ManagementRealm`, when the utility prompts for the realm name just accept the default unless you have switched to a different realm.

Interactive Mode

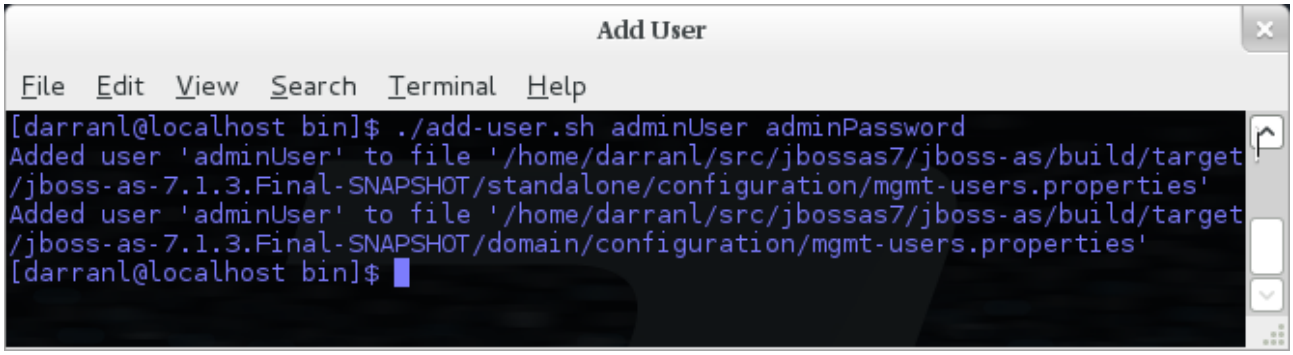


Here we have added a new Management User called `adminUser`, as you can see some of the questions offer default responses so you can just press enter without repeating the default value.

For now just answer `n` or `no` to the final question, adding users to be used by processes is described in more detail in the domain management chapter.

Interactive Mode

To add a user in non-interactive mode the command `./add-user.sh {username} {password}` can be used.

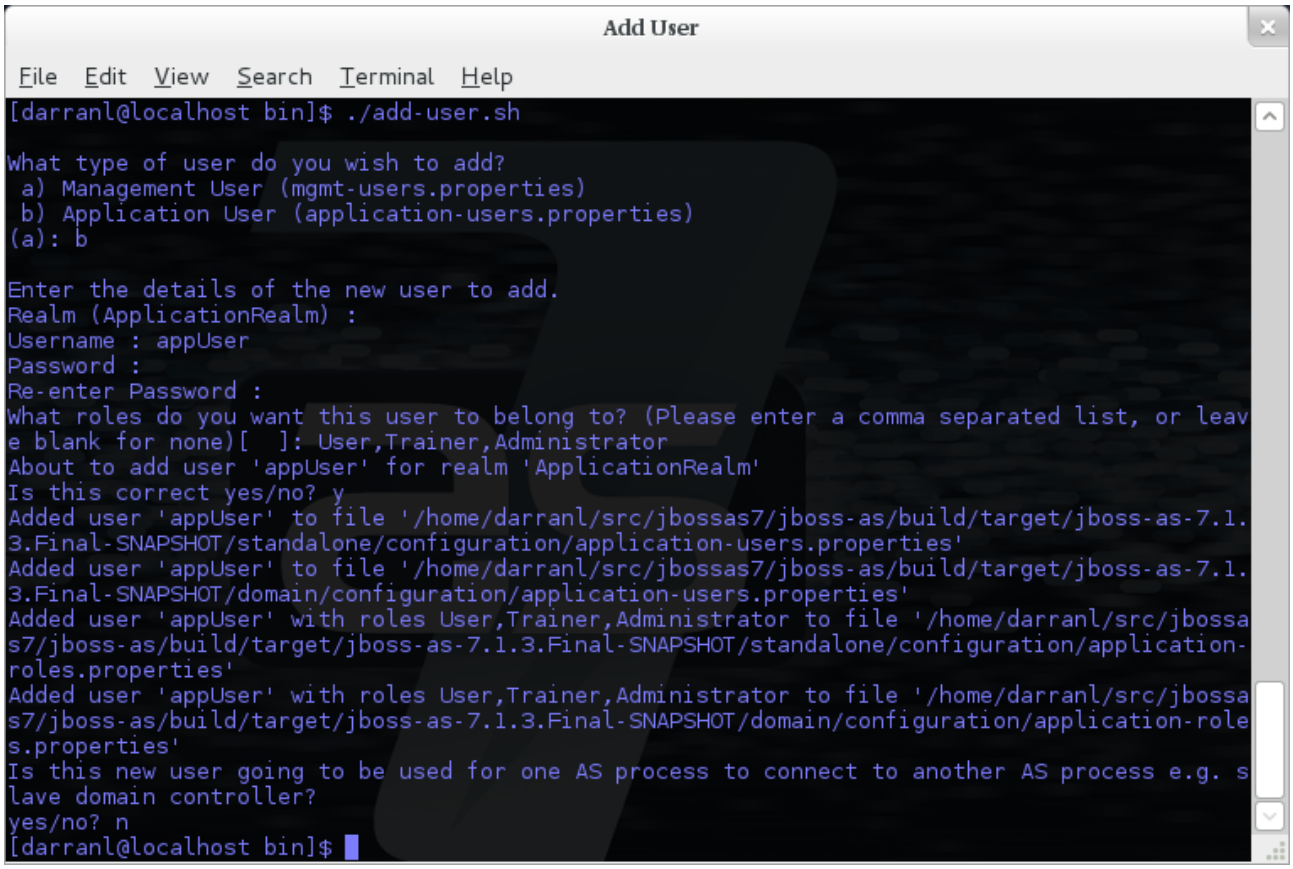


If you add users using this approach there is a risk that any other user that can view the list of running process may see the arguments including the password of the user being added, there is also the risk that the username / password combination will be cached in the history file of the shell you are currently using.

An Application User

When adding application users in addition to adding the user with their pre-hashed password it is also now possible to define the roles of the user.

Interactive Mode

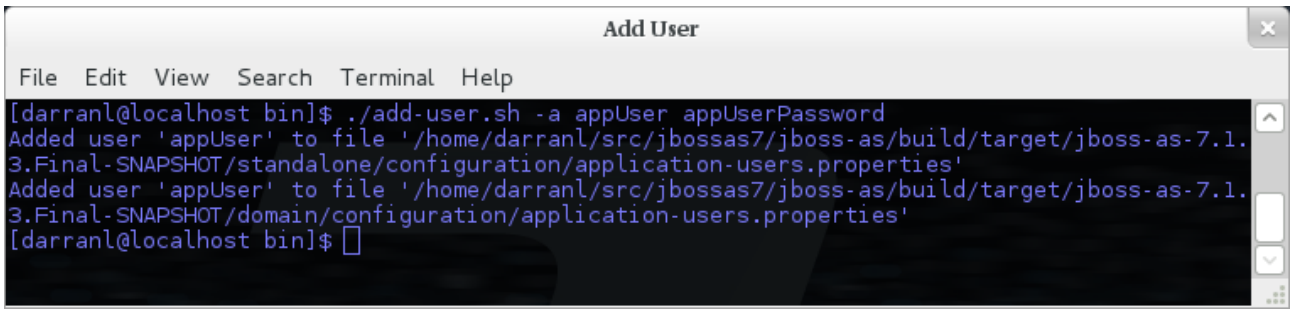


Here a new user called `appUser` has been added, in this case a comma separated list of roles has also been specified.

As with adding a management user just answer `n` or `no` to the final question until you know you are adding a user that will be establishing a connection from one server to another.

Interactive Mode

To add an application user non-interactively use the command `./add-user.sh -a {username} {password}`.



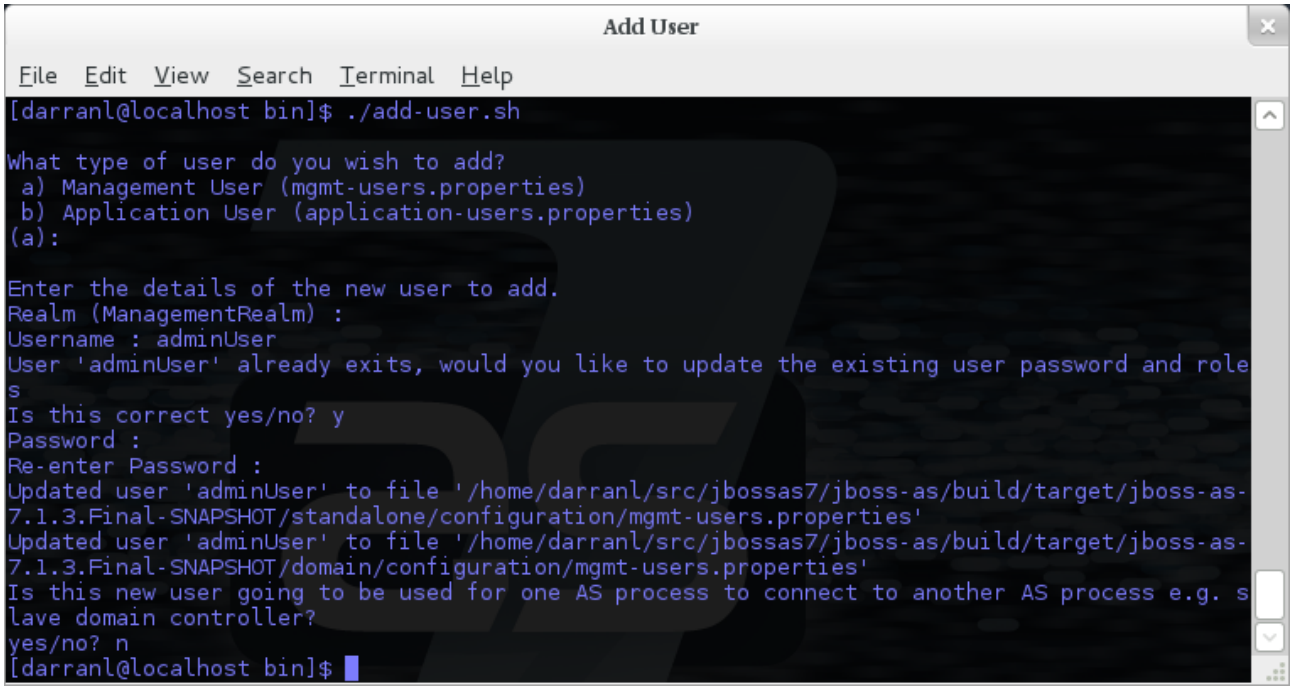
Non-interactive mode does not support defining a list of users, to associate a user with a set of roles you will need to manually edit the `application-roles.properties` file by hand.

Updating a User

Within the add-user utility it is also possible to update existing users, in interactive mode you will be prompted to confirm if this is your intention.

A Management User

Interactive Mode

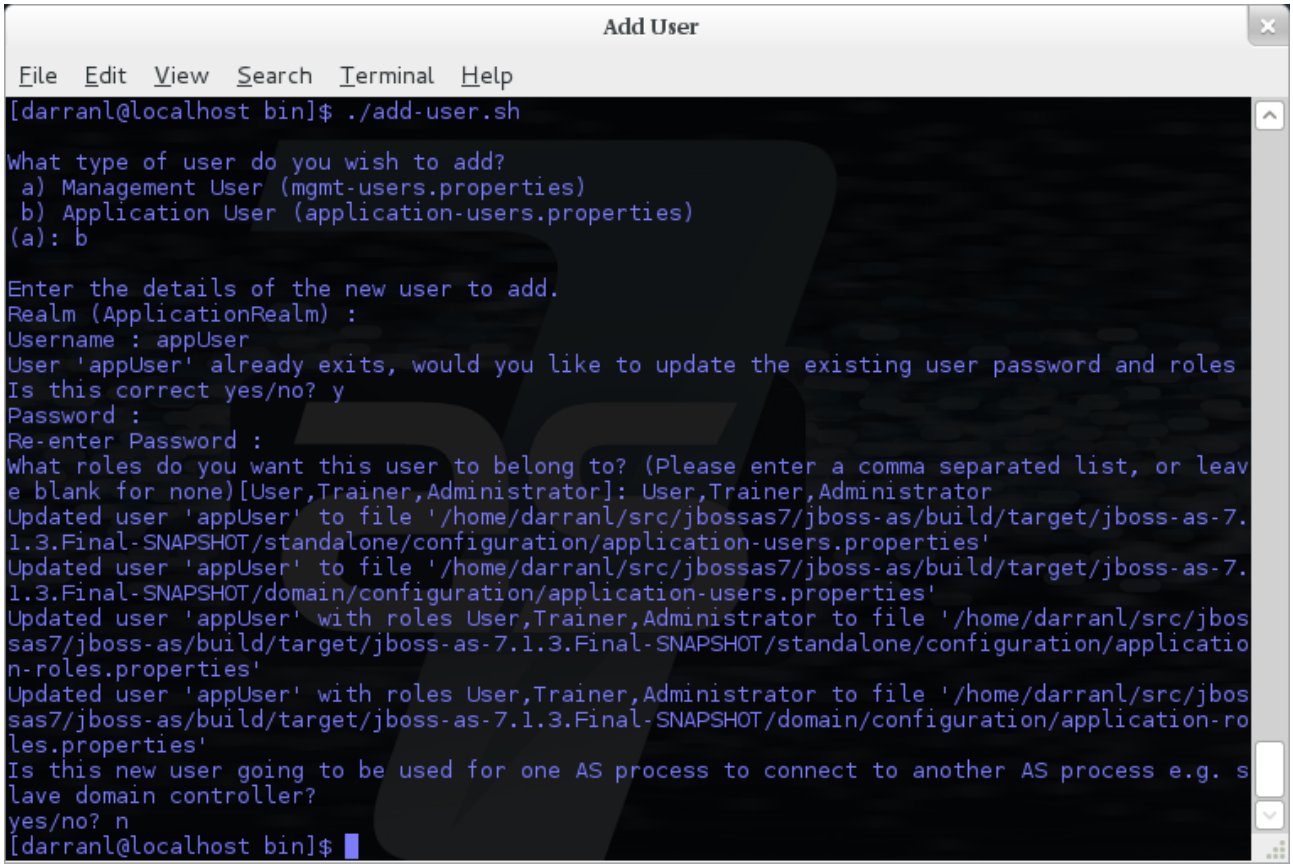


Interactive Mode

In non-interactive mode if a user already exists the update is automatic with no confirmation prompt.

An Application User

Interactive Mode



On updating a user with roles you will need to re-enter the list of roles assigned to the user.

Interactive Mode

In non-interactive mode if a user already exists the update is automatic with no confirmation prompt.

Community Contributions

There are still a few features to add to the add-user utility such as removing users or adding application users with roles in non-interactive mode, if you are interested in contributing to WildFly development the add-user utility is a good place to start as it is a stand alone utility, however it is a part of the AS build so you can become familiar with the AS development processes without needing to delve straight into the internals of the application server.

5.21.7 JMX Security

When configuring the security realms remote access to the server's MBeanServer needs a special mention. When running in standalone mode the following is the default configuration:

```
<subsystem xmlns="urn:jboss:domain:jmx:1.3">
  ...
  <remoting-connector/>
</subsystem>
```

With this configuration remote access to JMX is provided over the http management interface, this is secured using the realm `ManagementRealm`, this means that any user that can connect to the native interface can also use this interface to access the MBeanServer - to disable this just remove the `<remoting-connector />` element.

In domain mode it is slightly more complicates as the native interface is exposed by the host controller process however each application server is running in it's own process so by default remote access to JMX is disabled.

```
<subsystem xmlns="urn:jboss:domain:remoting:3.0">
  <http-connector name="http-remoting-connector" connector-ref="default"
security-realm="ApplicationRealm"/>
</subsystem>
```

```
<subsystem xmlns="urn:jboss:domain:jmx:1.3">
  ...
  <!--<remoting-connector use-management-endpoint="false"/>-->
</subsystem>
```

To enable remote access to JMX uncomment the `<remoting-connector />` element however be aware that this will make the MBeanServer accessible over the same Remoting connector used for remote JNDI and EJB access - this means that any user that can authenticate against the realm `ApplicationRealm` will be able to access the MBeanServer.

The following Jira issue is currently outstanding to allow access to the individual MBeanServers by proxying through the host controllers native interface AS7-4009, if this is a feature you would use please add your vote to the issue.

5.21.8 Detailed Configuration

This section of the documentation describes the various configuration options when defining realms, plug-ins are a slightly special case so the configuration options for plug-ins is within it's own section.

Within a security realm definition there are four optional elements `<plug-ins />`, `<server-identities />`, `<authentication />`, and `<authorization />`, as mentioned above plug-ins is defined within it's own section below so we will begin by looking at the `<server-identities />` element.

<server-identities />

The server identities section of a realm definition is used to define how a server appears to the outside world, currently this element can be used to configure a password to be used when establishing a remote outbound connection and also how to load a X.509 key which can be used for both inbound and outbound SSL connections.

<ssl />

```
<server-identities>
  <ssl protocol="...">
    <keystore path="..." relative-to="..." keystore-password="..." alias="..."
key-password="..." />
  </ssl>
</server-identities>
```

- **protocol** - By default this is set to TLS and in general does not need to be set.

The SSL element then contains the nested `<keystore />` element, this is used to define how to load the key from the file based (JKS) keystore.

- **path** (mandatory) - This is the path to the keystore, this can be an absolute path or relative to the next attribute.
- **relative-to** (optional) - The name of a service representing a path the keystore is relative to.
- **keystore-password** (mandatory) - The password required to open the keystore.
- **alias** (optional) - The alias of the entry to use from the keystore - for a keystore with multiple entries in practice the first usable entry is used but this should not be relied on and the alias should be set to guarantee which entry is used.
- **key-password** (optional) - The password to load the key entry, if omitted the keystore-password will be used instead.

If you see the error `UnrecoverableKeyException: Cannot recover key` the most likely cause that you need to specify a `key-password` and possible even an `alias` as well to ensure only one key is loaded.

<secret />

```
<server-identities>
  <secret value="..." />
</server-identities>
```

- **value** (mandatory) - The password to use for outbound connections encoded as Base64, this field also supports a vault expression should stronger protection be required.

The username for the outbound connection is specified at the point the outbound connection is defined.

<authentication />

The authentication element is predominantly used to configure the authentication that is performed on an inbound connection, however there is one exception and that is if a trust store is defined - on negotiating an outbound SSL connection the trust store will be used to verify the remote server.

```
<authentication>
  <truststore />
  <local />
  <jaas />
  <ldap />
  <properties />
  <users />
  <plug-in />
</authentication>
```

An authentication definition can have zero or one `<truststore />`, it can also have zero or one `<local />` and it can also have one of `<jaas />`, `<ldap />`, `<properties />`, `<users />`, and `<plug-in />` i.e. the local mechanism and a truststore for certificate verification can be independent switched on and off and a single username / password store can be defined.

<truststore />

```
<authentication>
  <truststore path="..." relative-to="..." keystore-password="..."/>
</authentication>
```

This element is used to define how to load a key store file that can be used as the trust store within the SSLContext we create internally, the store is then used to verify the certificates of the remote side of the connection be that inbound or outbound.

- **path** (mandatory) - This is the path to the keystore, this can be an absolute path or relative to the next attribute.
- **relative-to** (optional) - The name of a service representing a path the keystore is relative to.
- **keystore-password** (mandatory) - The password required to open the keystore.

Although this is a definition of a trust store the attribute for the password is `keystore-password`, this is because the underlying file being opened is still a key store.

<local />

```
<authentication>
  <local default-user="..." allowed-users="..." />
</authentication>
```

This element switches on the local authentication mechanism that allows clients to the server to verify that they are local to the server, at the protocol level it is optional for the remote client to send a user name in the authentication response.

- **default-user** (optional) - If the client does not pass in a username this is the assumed username, this value is also automatically added to the list of allowed-users.
- **allowed-users** (optional) - This attribute is used to specify a comma separated list of users allowed to authenticate using the local mechanism, alternatively '*' can be specified to allow any username to be specified.

<jaas />

```
<authentication>
  <jaas name="..." />
</authentication>
```

The jaas element is used to enable username and password based authentication where the supplied username and password are verified by making use of a configured jaas domain.

- **name** (mandatory) - The name of the jaas domain to use to verify the supplied username and password.

As JAAS authentication works by taking a username and password and verifying these the use of this element means that at the transport level authentication will be forced to send the password in plain text, any interception of the messages exchanged between the client and server without SSL enabled will reveal the users password.

<ldap />

```
<authentication>
  <ldap connection="..." base-dn="..." recursive="..." user-dn="...">
    <username-filter attribute="..." />
    <advanced-filter filter="..." />
  </ldap>
</authentication>
```

The ldap element is used to define how LDAP searches will be used to authenticate a user, this works by first connecting to LDAP and performing a search using the supplied user name to identity the distinguished name of the user and then a subsequent connection is made to the server using the password supplied by the user - if this second connection is a success then authentication succeeds.

Due to the verification approach used this configuration causes the authentication mechanisms selected for the protocol to cause the password to be sent from the client in plain text, the following Jira issue is to investigating proxying a Digest authentication with the LDAP server so no plain text password is needed AS7-4195.

- **connection** (mandatory) - The name of the connection to use to connect to LDAP.
- **base-dn** (mandatory) - The distinguished name of the context to use to begin the search from.
- **recursive** (optional) - Should the filter be executed recursively? Defaults to false.
- **user-dn** (optional) - After the user has been found specifies which attribute to read for the users distinguished name, defaults to 'dn'.

Within the ldap element only one of `<username-filter />` or `<advanced-filter />` can be specified.

<username-filter />

This element is used for a simple filter to match the username specified by the remote user against a single attribute, as an example with Active Directory the match is most likely to be against the 'sAMAccountName' attribute.

- **attribute** (mandatory) - The name of the field to match the users supplied username against.

<advanced-filter />

This element is used where a more advanced filter is required, one example use of this filter is to exclude certain matches by specifying some additional criteria for the filter.

- **filter** (mandatory) - The filter to execute to locate the user, this filter should contain '{0}' as a place holder for the username supplied by the user authenticating.

<properties />

```
<authentication>
  <properties path="..." relative-to="..." plain-text="..." />
</authentication>
```

The properties element is used to reference a properties file to load to read a users password or pre-prepared digest for the authentication process.

- **path** (mandatory) - The path to the properties file, either absolute or relative to the path referenced by the relative-to attribute.
- **relative-to** (optional) - The name of a path service that the defined path will be relative to.
- **plain-text** (optional) - Setting to specify if the passwords are stored as plain text within the properties file, defaults to false.

By default the properties files are expected to store a pre-prepared hash of the users password in the form HEX(MD5(username ':' realm ':' password))

<users />

```
<authentication>
  <users>
    <user username="...">
      <password>...</password>
    </user>
  </users>
</authentication>
```

This is a very simple store of a username and password that stores both of these within the domain model, this is only really provided for the provision of simple examples.

- **username** (mandatory) - A users username.

The `<password/>` element is then used to define the password for the user.

<authorization />

The authorization element is used to define how a users roles can be loaded after the authentication process completes, these roles may then be used for subsequent authorization decisions based on the service being accessed. At the moment only a properties file approach or a custom plug-in are supported - support for loading roles from LDAP or from a database are planned for a subsequent release.

```
<authorization>
  <properties />
  <plug-in />
</authorization>
```

<properties />

```
<authorization>
  <properties path="..." relative-to="..." />
</authorization>
```

The format of the properties file is `username={ROLES}` where {ROLES} is a comma separated list of the users roles.

- **path** (mandatory) - The path to the properties file, either absolute or relative to the path referenced by the relative-to attribute.
- **relative-to** (optional) - The name of a path service that the defined path will be relative to.

<outbound-connection />

Strictly speaking these are not a part of the security realm definition, however at the moment they are only used by security realms so the definition of outbound connection is described here.

```
<management>
  <security-realms />
  <outbound-connections>
    <ldap />
  </outbound-connections>
</management>
```

<ldap />

At the moment we only support outbound connections to ldap servers for the authentication process - this will later be expanded when we add support for database based authentication.

```
<outbound-connections>
  <ldap name="..." url="..." search-dn="..." search-credential="..."
initial-context-factory="..." />
</outbound-connections>
```

The outbound connections are defined in this section and then referenced by name from the configuration that makes use of them.

- **name** (mandatory) - The unique name used to reference this connection.
- **url** (mandatory) - The URL use to establish the LDAP connection.
- **search-dn** (mandatory) - The distinguished name of the user to authenticate as to perform the searches.
- **search-credential** (mandatory) - The password required to connect to LDAP as the search-dn.
- **initial-context-factory** (optional) - Allows overriding the initial context factory, defaults to '`com.sun.jndi.ldap.LdapCtxFactory`'

5.21.9 Plug Ins

Within WildFly 8 for communication with the management interfaces and for other services exposed using Remoting where username / password authentication is used the use of Digest authentication is preferred over the use of HTTP Basic or SASL Plain so that we can avoid the sending of password in the clear over the network. For validation of the digests to work on the server we either need to be able to retrieve a users plain text password or we need to be able to obtain a ready prepared hash of their password along with the username and realm.

Previously to allow the addition of custom user stores we have added an option to the realms to call out to a JAAS domain to validate a users username and password, the problem with this approach is that to call JAAS we need the remote user to send in their plain text username and password so that a JAAS LoginModule can perform the validation, this forces us down to use either the HTTP Basic authentication mechanism or the SASL Plain mechanism depending on the transport used which is undesirable as we can not longer use Digest.

To overcome this we now support plugging in custom user stores to support loading a users password, hash and roles from a custom store to allow different stores to be implemented without forcing the authentication back to plain text variant, this article describes the requirements for a plug in and shows a simple example plug-in for use with WildFly 8.

When implementing a plug in there are two steps to the authentication process, the first step is to load the users identity and credential from the relevant store - this is then used to verify the user attempting to connect is valid. After the remote user is validated we then load the users roles in a second step. For this reason the support for plug-ins is split into the two stages, when providing a plug-in either of these two steps can be implemented but there is no requirement to implement the other side.

When implementing a plug-in the following interfaces are the bare minimum that need to be implemented so depending on if a plug-in to load a users identity or a plug-in to load a users roles is being implemented you will be implementing one of these interfaces.

Note *- All classes and interfaces of the SPI to be implemented are in the 'org.jboss.as.domain.management.plugin' package which is a part of the 'org.jboss.as.domain-management' module but for simplicity for the rest of this section only the short names will be shown.*

AuthenticationPlugIn

To implement an `AuthenticationPlugIn` the following interface needs to be implemened: -

```
public interface AuthenticationPlugIn<T extends Credential> {
    Identity<T> loadIdentity(final String userName, final String realm) throws IOException;
}
```

During the authentication process this method will be called with the user name supplied by the remote user and the name of the realm they are authenticating against, this method call represents that an authentication attempt is occurring but it is the Identity instance that is returned that will be used for the actual authentication to verify the remote user.

The Identity interface is also an interface you will implement: -

```
public interface Identity<T extends Credential> {
    String getUserName();
    T getCredential();
}
```

Additional information can be contained within the Identity implementation although it will not currently be used, the key piece of information here is the Credential that will be returned - this needs to be one of the following: -

PasswordCredential

```
public final class PasswordCredential implements Credential {
    public PasswordCredential(final char[] password);
    public char[] getPassword();
    void clear();
}
```

The `PasswordCredential` is already implemented so use this class if you have the plain text password of the remote user, by using this the secured interfaces will be able to continue using the Digest mechanism for authentication.

DigestCredential

```
public final class DigestCredential implements Credential {
    public DigestCredential(final String hash);
    public String getHash();
}
```

This class is also already implemented and should be returned if instead of the plain text password you already have a pre-prepared hash of the username, realm and password.

ValidatePasswordCredential

```
public interface ValidatePasswordCredential extends Credential {
    boolean validatePassword(final char[] password);
}
```

This is a special Credential type to use when it is not possible to obtain either a plain text representation of the password or a pre-prepared hash - this is an interface as you will need to provide an implementation to verify a supplied password. The down side of using this type of Credential is that the authentication mechanism used at the transport level will need to drop down from Digest to either HTTP Basic or SASL Plain which will now mean that the remote client is sending their credential across the network in the clear.

If you use this type of credential be sure to force the mechanism choice to Plain as described in the configuration section below.

AuthorizationPlugIn

If you are implementing a custom mechanism to load a users roles you need to implement the `AuthorizationPlugIn`

```
public interface AuthorizationPlugIn {
    String[] loadRoles(final String userName, final String realm) throws IOException;
}
```

As with the `AuthenticationPlugIn` this has a single method that takes a users userName and realm - the return type is an array of Strings with each entry representing a role the user is a member of.

PlugInConfigurationSupport

In addition to the specific interfaces above there is an additional interface that a plug-in can implement to receive configuration information before the plug-in is used and also to receive a Map instance that can be used to share state between the plug-in instance used for the authentication step of the call and the plug-in instance used for the authorization step.

```
public interface PlugInConfigurationSupport {
    void init(final Map<String, String> configuration, final Map<String, Object> sharedState)
throws IOException;
}
```

Installing and Configuring a Plug-In

The next step of this article describes the steps to implement a plug-in provider and how to make it available within WildFly 8 and how to configure it. Example configuration and an example implementation are shown to illustrate this.

The following is an example security realm definition which will be used to illustrate this: -

```
<security-realm name="PlugInRealm">
   <plug-ins>
      <plug-in module="org.jboss.as.sample.plugin"/>
   </plug-ins>
   <authentication>
      <plug-in name="Sample">
         <properties>
            <property name="darranl.password" value="dpd"/>
            <property name="darranl.roles" value="Admin,Banker,User"/>
         </properties>
      </plug-in>
   </authentication>
   <authorization>
      <plug-in name="Delegate" />
   </authorization>
</security-realm>
```

Before looking closely at the packaging and configuration there is one more interface to implement and that is the `PlugInProvider` interface, that interface is responsible for making PlugIn instances available at runtime to handle the requests.

PlugInProvider

```
public interface PlugInProvider {
    AuthenticationPlugIn<Credential> loadAuthenticationPlugIn(final String name);
    AuthorizationPlugIn loadAuthorizationPlugIn(final String name);
}
```

These methods are called with the name that is supplied in the plug-in elements that are contained within the authentication and authorization elements of the configuration, based on the sample configuration above the loadAuthenticationPlugIn method will be called with a parameter of 'Sample' and the loadAuthorizationPlugIn method will be called with a parameter of 'Delegate'.

Multiple plug-in providers may be available to the application server so if a `PlugInProvider` implementation does not recognise a name then it should just return null and the server will continue searching the other providers. If a `PlugInProvider` does recognise a name but fails to instantiate the PlugIn then a `RuntimeException` can be thrown to indicate the failure.

As a server could have many providers registered it is recommended that a naming convention including some form of hierarchy is used e.g. use package style names to avoid conflicts.

For the example the implementation is as follows: -

```
public class SamplePluginProvider implements PlugInProvider {

    public AuthenticationPlugIn<Credential> loadAuthenticationPlugIn(String name) {
        if ("Sample".equals(name)) {
            return new SampleAuthenticationPlugIn();
        }
        return null;
    }

    public AuthorizationPlugIn loadAuthorizationPlugIn(String name) {
        if ("Sample".equals(name)) {
            return new SampleAuthenticationPlugIn();
        } else if ("Delegate".equals(name)) {
            return new DelegateAuthorizationPlugIn();
        }
        return null;
    }
}
```

The load methods are called for each authentication attempt but it will be an implementation detail of the provider if it decides to return a new instance of the provider each time - in this scenario as we also use configuration and shared state then new instances of the implementations make sense.

To load the provider use a ServiceLoader so within the META-INF/services folder of the jar this project adds a file called '`org.jboss.as.domain.management.plugin.PlugInProvider`' - this contains a single entry which is the fully qualified class name of the PlugInProvider implementation class.

```
org.jboss.as.sample.SamplePluginProvider
```

Package as a Module

To make the `PlugInProvider` available to the application it is bundled as a module and added to the modules already shipped with WildFly 8.

To add as a module we first need a `module.xml`: -

```
<?xml version="1.0" encoding="UTF-8"?>

<module xmlns="urn:jboss:module:1.1" name="org.jboss.as.sample.plugin">
    <properties>
    </properties>

    <resources>
        <resource-root path="SamplePlugIn.jar"/>
    </resources>

    <dependencies>
        <module name="org.jboss.as.domain-management" />
    </dependencies>
</module>
```

The interfaces being implemented are in the '`org.jboss.as.domain-management`' module so a dependency on that module is defined, this `module.xml` is then placed in the '{ `jboss.home}/modules/org/jboss/as/sample/plugin/main`'.

The compiled classed and `META-INF/services` as described above are assembled into a jar called `SamplePlugIn.jar` and also placed into this folder.

Looking back at the sample configuration at the top of the realm definition the following element was added: -

```
<plug-ins>
      <plug-in module="org.jboss.as.sample.plugin"/>
   </plug-ins>
```

This element is used to list the modules that should be searched for plug-ins. As plug-ins are loaded during the server start up this search is a lazy search so don't expect a definition to a non existant module or to a module that does not contain a plug-in to report an error.

The AuthenticationPlugIn

The example `AuthenticationPlugIn` is implemented as: -

```
public class SampleAuthenticationPlugIn extends AbstractPlugIn {

    private static final String PASSWORD_SUFFIX = ".password";
    private static final String ROLES_SUFFIX = ".roles";
    private Map<String, String> configuration;

    public void init(Map<String, String> configuration, Map<String, Object> sharedState) throws
IOException {
        this.configuration = configuration;
        // This will allow an AuthorizationPlugIn to delegate back to this instance.
        sharedState.put(AuthorizationPlugIn.class.getName(), this);
    }

    public Identity loadIdentity(String userName, String realm) throws IOException {
        String passwordKey = userName + PASSWORD_SUFFIX;
        if (configuration.containsKey(passwordKey)) {
            return new SampleIdentity(userName, configuration.get(passwordKey));
        }
        throw new IOException("Identity not found.");
    }

    public String[] loadRoles(String userName, String realm) throws IOException {
        String rolesKey = userName + ROLES_SUFFIX;
        if (configuration.containsKey(rolesKey)) {
            String roles = configuration.get(rolesKey);
            return roles.split(",");
        } else {
            return new String[0];
        }
    }

    private static class SampleIdentity implements Identity {
        private final String userName;
        private final Credential credential;

        private SampleIdentity(final String userName, final String password) {
            this.userName = userName;
            this.credential = new PasswordCredential(password.toCharArray());
        }

        public String getUserName() {
            return userName;
        }

        public Credential getCredential() {
            return credential;
        }
    }
}
```

As you can see from this implementation there is also an additional class being extended `AbstractPlugIn` - that is simply an abstract class that implements the `AuthenticationPlugIn`, `AuthorizationPlugIn`, and `PlugInConfigurationSupport` interfaces already. The properties that were defined in the configuration are passed in as a Map and importantly for this sample the plug-in adds itself to the shared state map.

The AuthorizationPlugIn

The example implementation of the authentication plug in is as follows: -

```
public class DelegateAuthorizationPlugIn extends AbstractPlugIn {

    private AuthorizationPlugIn authorizationPlugIn;

    public void init(Map<String, String> configuration, Map<String, Object> sharedState) throws
IOException {
        authorizationPlugIn = (AuthorizationPlugIn)
sharedState.get(AuthorizationPlugIn.class.getName());
    }

    public String[] loadRoles(String userName, String realm) throws IOException {
        return authorizationPlugIn.loadRoles(userName, realm);
    }

}
```

This plug-in illustrates how two plug-ins can work together, by the `AuthenticationPlugIn` placing itself in the shared state map it is possible for the authorization plug-in to make use of it for the loadRoles implementation.

Another option to consider to achieve similar behaviour could be to provide an Identity implementation that also contains the roles and place this in the shared state map - the `AuthorizationPlugIn` can retrieve this and return the roles.

Forcing Plain Text Authentication

As mentioned earlier in this article if the `ValidatePasswordCredential` is going to be used then the authentication used at the transport level needs to be forced from Digest authentication to plain text authentication, this can be achieved by adding a mechanism attribute to the plug-in definition within the authentication element i.e.

```
<authentication>
    <plug-in name="Sample" mechanism="PLAIN">
```

5.21.10 Example Configurations

This section of the document contains a couple of examples for the most common scenarios likely to be used with the security realms, please feel free to raise Jira issues requesting additional scenarios or if you have configured something not covered here please feel free to add your own examples - this document is editable after all

At the moment these examples are making use of the '`ManagementRealm`' however the same can apply to the '`ApplicationRealm`' or any custom realm you create for yourselves.

LDAP Authentication

The following example demonstrates an example configuration making use of Active Directory to verify the users username and password.

```
<management>
  <security-realms>
    <security-realm name="ManagementRealm">
      <authentication>
        <ldap connection="EC2" base-dn="CN=Users,DC=darranl,DC=jboss,DC=org">
          <username-filter attribute="sAMAccountName" />
        </ldap>
      </authentication>
    </security-realm>

  </security-realms>

  <outbound-connections>
    <ldap name="EC2" url="ldap://127.0.0.1:9797"
search-dn="CN=wf8,CN=Users,DC=darranl,DC=jboss,DC=org" search-credential="password"/>
  </outbound-connections>

  ...

</management>
```

> For simplicity the `<local/>` configuration has been removed from this example, however there it is fine to leave that in place for local authentication to remain possible.

Enable SSL

The first step is the creation of the key, by default this is going to be used for both the native management interface and the http management interface - to create the key we can use the `keyTool`, the following example will create a key valid for one year.

Open a terminal window in the folder `{jboss.home}/standalone/configuration` and enter the following command: -

`keytool -genkey -alias server -keyalg RSA -keystore server.keystore -validity 365`

```
Enter keystore password:
Re-enter new password:
```

In this example I choose '`keystore_password`'.

```
What is your first and last name?
  [Unknown]:  localhost
```

Of all of the questions asked this is the most important and should match the host name that will be entered into the web browser to connect to the admin console.

Answer the remaining questions as you see fit and at the end for the purpose of this example I set the key password to '`key_password`'.

The following example shows how this newly created keystore will be referenced to enable SSL.

```
<security-realm name="ManagementRealm">
  <server-identities>
    <ssl>
      <keystore path="server.keystore" relative-to="jboss.server.config.dir"
keystore-password="keystore_password" alias="server" key-password="key_password" />
    </ssl>
  </server-identities>
  <authentication>
    ...
  </authentication>
</security-realm>
```

The contents of the `<authentication />` have not been changed in this example so authentication still occurs using either the local mechanism or username/password authentication using Digest.

Add Client-Cert to SSL

To enable Client-Cert style authentication we just now need to add a `<truststore />` element to the `<authentication />` element referencing a trust store that has had the certificates or trusted clients imported.

```
<security-realm name="ManagementRealm">
  <server-identities>
    <ssl>
      <keystore path="server.keystore" relative-to="jboss.server.config.dir"
keystore-password="keystore_password" alias="server" key-password="key_password" />
    </ssl>
  </server-identities>
  <authentication>
    <truststore path="server.truststore" relative-to="jboss.server.config.dir"
keystore-password="truststore_password" />
    <local default-user="$local"/>
    <properties path="mgmt-users.properties" relative-to="jboss.server.config.dir"/>
  </authentication>
</security-realm>
```

In this scenario if Client-Cert authentication does not occur clients can fall back to use either the local mechanism or username/password authentication. To make Client-Cert based authentication mandatory just remove the `<local />` and `<properties />` elements.

5.21.11 user utility

For use with the default configuration we supply a utility `add-user` which can be used to manage the properties files for the default realms used to store the users and their roles.

The add-user utility can be used to manage both the users in the `ManagementRealm` and the users in the `ApplicationRealm`, changes made apply to the properties file used both for domain mode and standalone mode.

After you have installed your application server and decided if you are going to run in standalone mode or domain mode you can delete the parent folder for the mode you are not using, the add-user utility will then only be managing the properties file for the mode in use.

The add-user utility is a command line utility however it can be run in both interactive and non-interactive mode. Depending on your platform the script to run the add-user utility is either `add-user.sh` or `add-user.bat` which can be found in `{jboss.home}/bin`.

This guide now contains a couple of examples of this utility in use to accomplish the most common tasks.

Adding a User

Adding users to the properties files is the primary purpose of this utility. Usernames can only contain the following characters in any number and in any order:

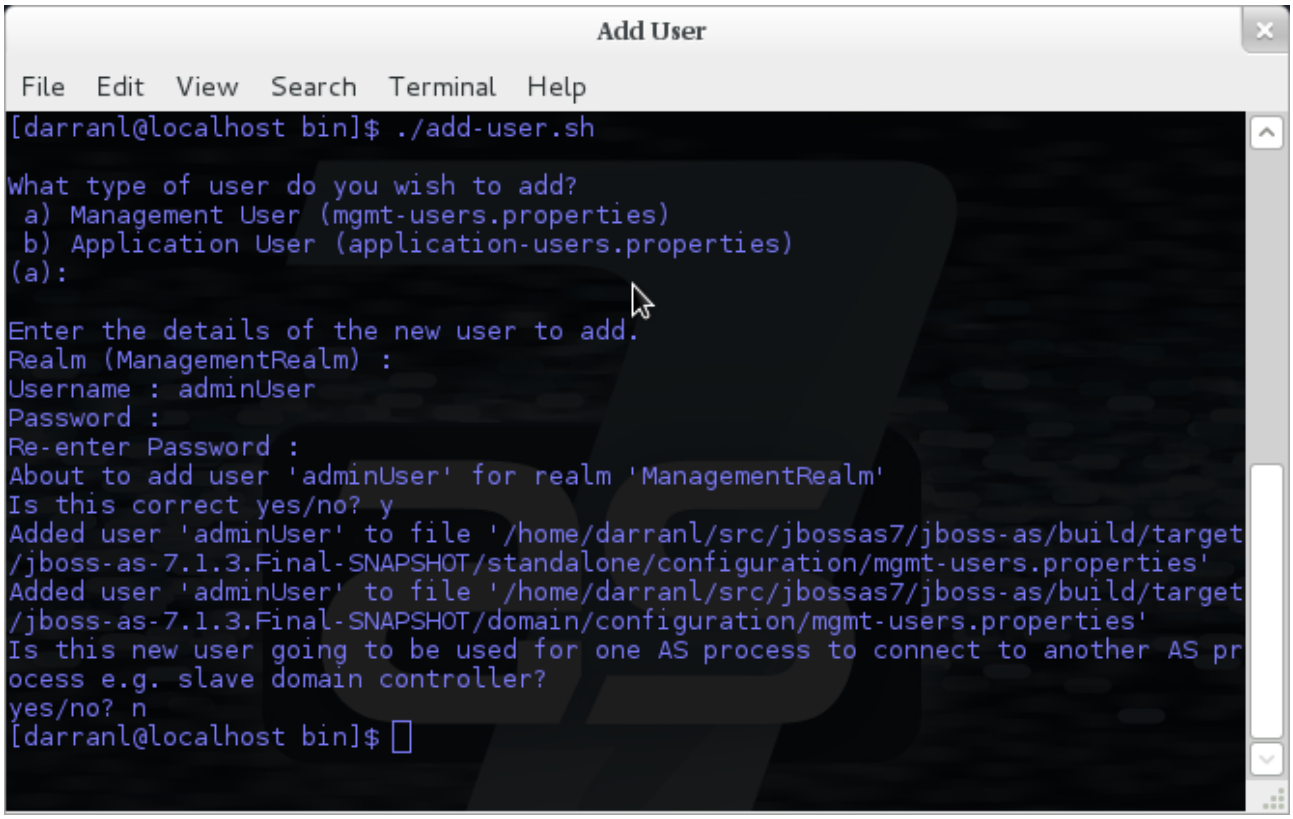
- Alphanumeric characters (a-z, A-Z, 0-9)
- Dashes (-), periods (.), commas (,), at (@)
- Escaped backslash (\\ )
- Escaped equals (\=)

The server caches the contents of the properties files in memory, however the server does check the modified time of the properties files on each authentication request and re-load if the time has been updated - this means all changes made by this utility are immediately applied to any running server.

A Management User

The default name of the realm for management users is `ManagementRealm`, when the utility prompts for the realm name just accept the default unless you have switched to a different realm.

Interactive Mode

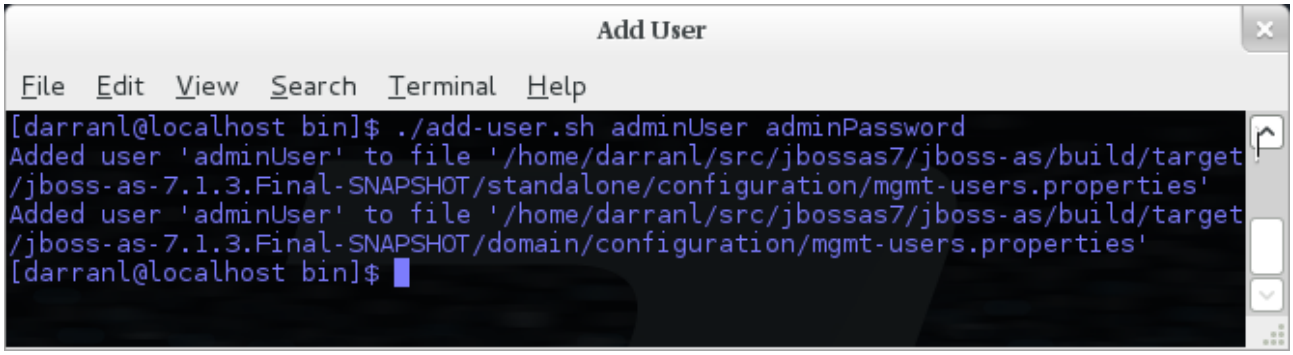


Here we have added a new Management User called `adminUser`, as you can see some of the questions offer default responses so you can just press enter without repeating the default value.

For now just answer `n` or `no` to the final question, adding users to be used by processes is described in more detail in the domain management chapter.

Interactive Mode

To add a user in non-interactive mode the command `./add-user.sh {username} {password}` can be used.

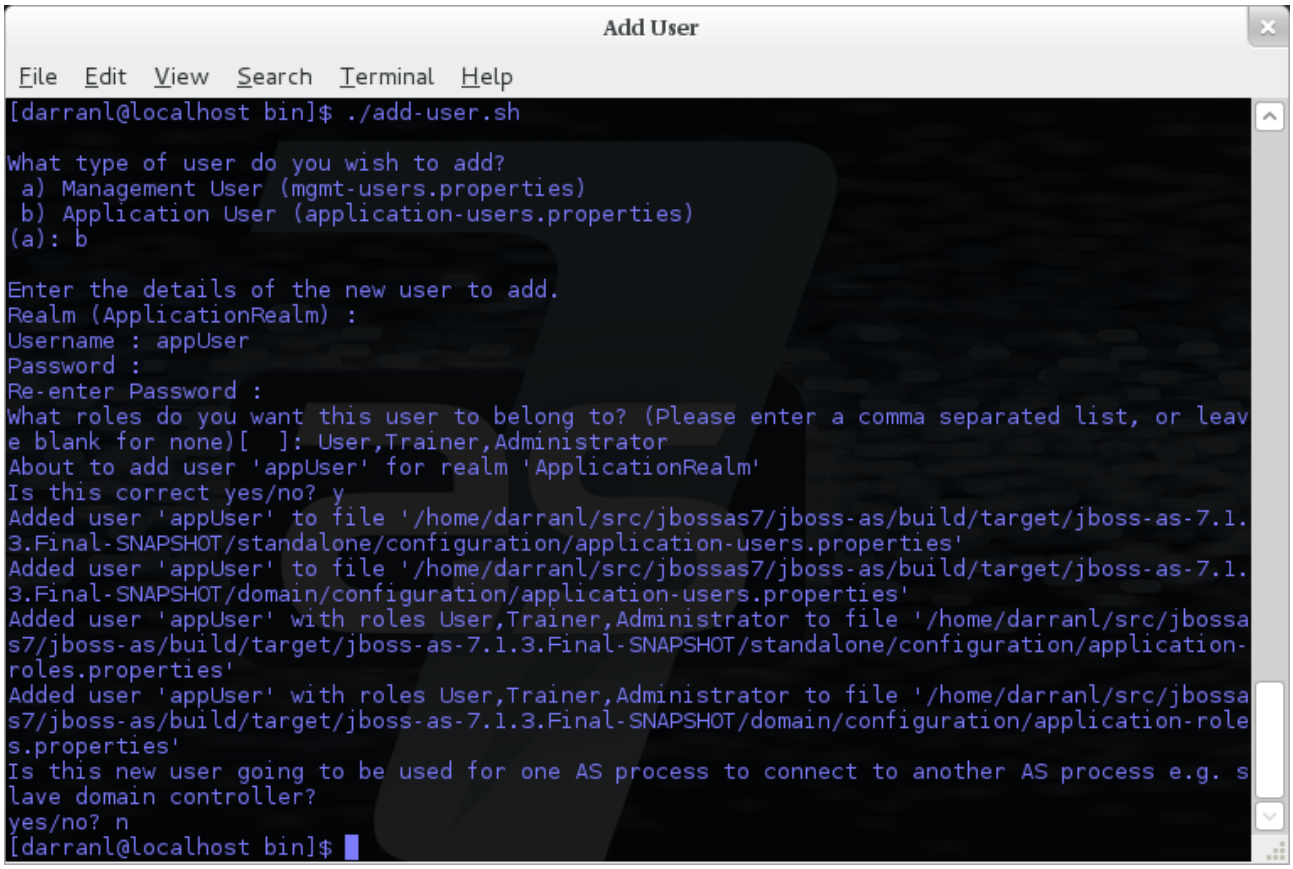


If you add users using this approach there is a risk that any other user that can view the list of running process may see the arguments including the password of the user being added, there is also the risk that the username / password combination will be cached in the history file of the shell you are currently using.

An Application User

When adding application users in addition to adding the user with their pre-hashed password it is also now possible to define the roles of the user.

Interactive Mode

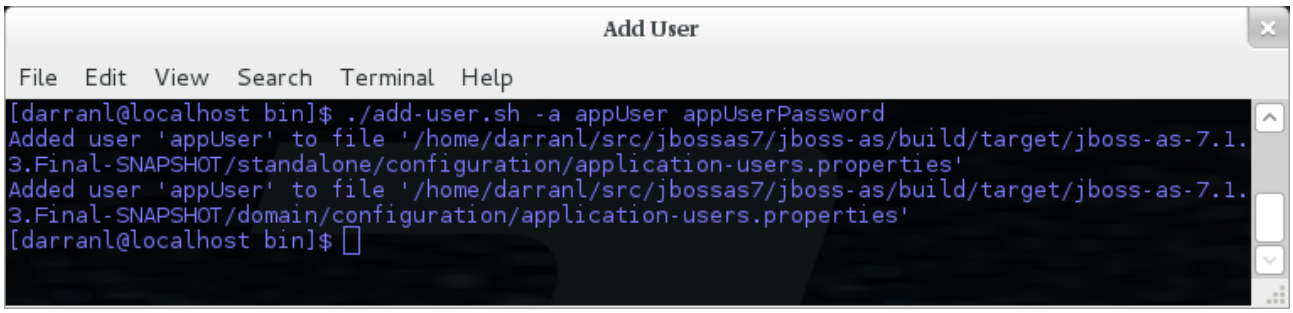


Here a new user called `appUser` has been added, in this case a comma separated list of roles has also been specified.

As with adding a management user just answer `n` or `no` to the final question until you know you are adding a user that will be establishing a connection from one server to another.

Interactive Mode

To add an application user non-interactively use the command `./add-user.sh -a {username} {password}`.



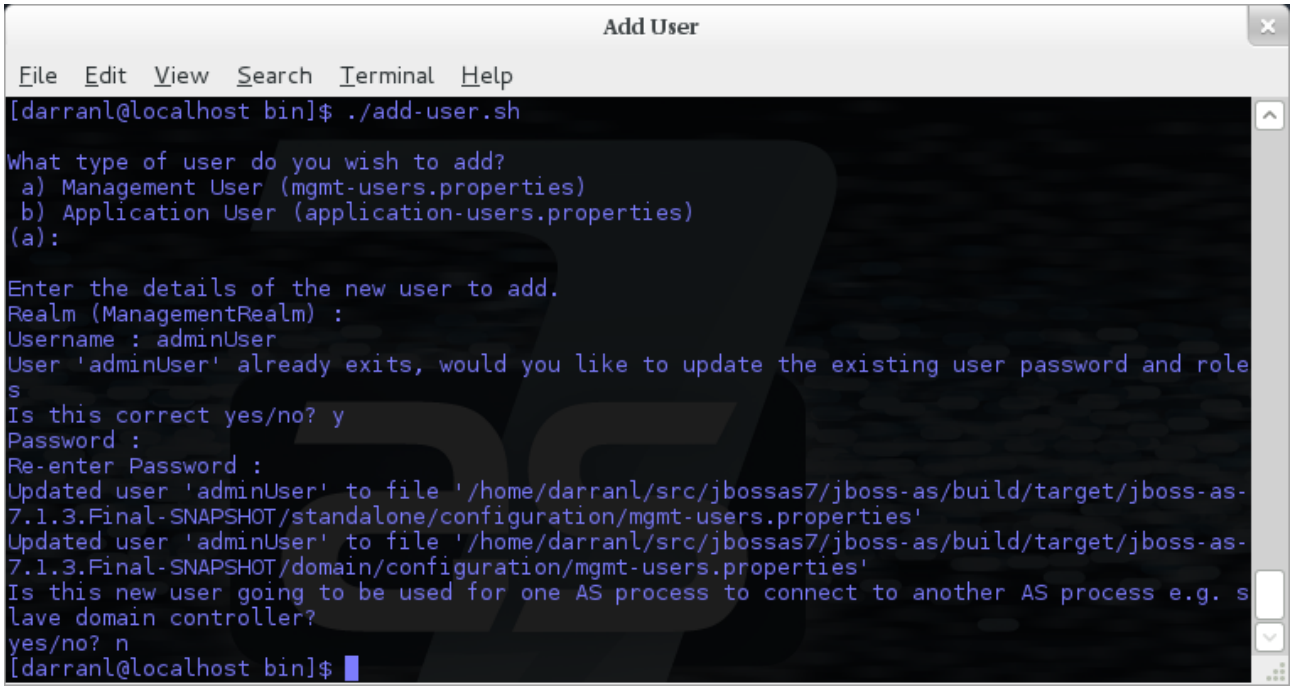
Non-interactive mode does not support defining a list of users, to associate a user with a set of roles you will need to manually edit the `application-roles.properties` file by hand.

Updating a User

Within the add-user utility it is also possible to update existing users, in interactive mode you will be prompted to confirm if this is your intention.

A Management User

Interactive Mode

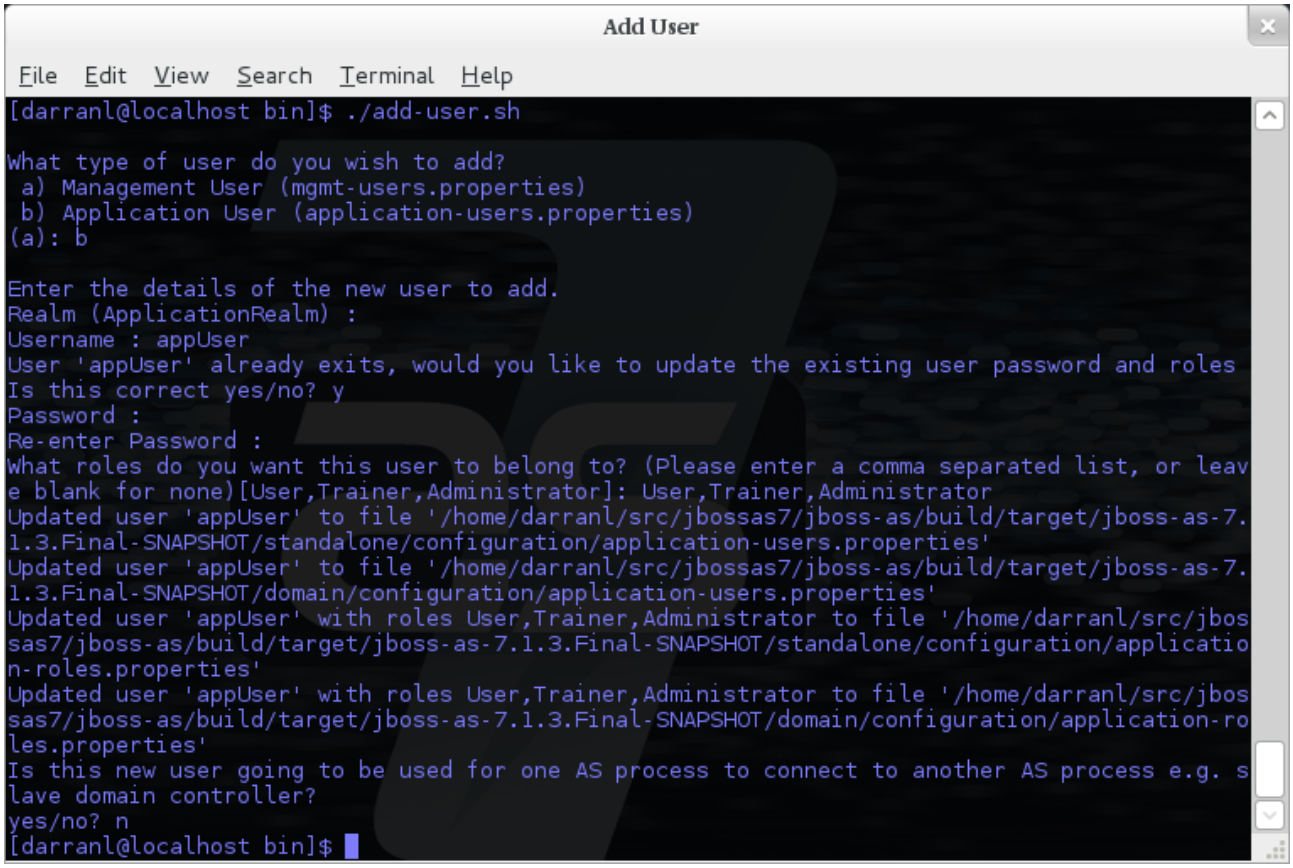


Interactive Mode

In non-interactive mode if a user already exists the update is automatic with no confirmation prompt.

An Application User

Interactive Mode



On updating a user with roles you will need to re-enter the list of roles assigned to the user.

Interactive Mode

In non-interactive mode if a user already exists the update is automatic with no confirmation prompt.

Community Contributions

There are still a few features to add to the add-user utility such as removing users or adding application users with roles in non-interactive mode, if you are interested in contributing to WildFly development the add-user utility is a good place to start as it is a stand alone utility, however it is a part of the AS build so you can become familiar with the AS development processes without needing to delve straight into the internals of the application server.

5.21.12 Detailed Configuration

This section of the documentation describes the various configuration options when defining realms, plug-ins are a slightly special case so the configuration options for plug-ins is within it's own section.

Within a security realm definition there are four optional elements `<plug-ins />`, `<server-identities />`, `<authentication />`, and `<authorization />`, as mentioned above plug-ins is defined within it's own section below so we will begin by looking at the `<server-identities />` element.

<server-identities />

The server identities section of a realm definition is used to define how a server appears to the outside world, currently this element can be used to configure a password to be used when establishing a remote outbound connection and also how to load a X.509 key which can be used for both inbound and outbound SSL connections.

<ssl />

```
<server-identities>
  <ssl protocol="...">
    <keystore path="..." relative-to="..." keystore-password="..." alias="..."
key-password="..." />
  </ssl>
</server-identities>
```

- **protocol** - By default this is set to TLS and in general does not need to be set.

The SSL element then contains the nested `<keystore />` element, this is used to define how to load the key from the file based (JKS) keystore.

- **path** (mandatory) - This is the path to the keystore, this can be an absolute path or relative to the next attribute.
- **relative-to** (optional) - The name of a service representing a path the keystore is relative to.
- **keystore-password** (mandatory) - The password required to open the keystore.
- **alias** (optional) - The alias of the entry to use from the keystore - for a keystore with multiple entries in practice the first usable entry is used but this should not be relied on and the alias should be set to guarantee which entry is used.
- **key-password** (optional) - The password to load the key entry, if omitted the keystore-password will be used instead.

If you see the error `UnrecoverableKeyException: Cannot recover key` the most likely cause that you need to specify a `key-password` and possible even an `alias` as well to ensure only one key is loaded.

<secret />

```
<server-identities>
  <secret value="..." />
</server-identities>
```

- **value** (mandatory) - The password to use for outbound connections encoded as Base64, this field also supports a vault expression should stronger protection be required.

The username for the outbound connection is specified at the point the outbound connection is defined.

<authentication />

The authentication element is predominantly used to configure the authentication that is performed on an inbound connection, however there is one exception and that is if a trust store is defined - on negotiating an outbound SSL connection the trust store will be used to verify the remote server.

```
<authentication>
  <truststore />
  <local />
  <jaas />
  <ldap />
  <properties />
  <users />
  <plug-in />
</authentication>
```

An authentication definition can have zero or one `<truststore />`, it can also have zero or one `<local />` and it can also have one of `<jaas />`, `<ldap />`, `<properties />`, `<users />`, and `<plug-in />` i.e. the local mechanism and a truststore for certificate verification can be independent switched on and off and a single username / password store can be defined.

<truststore />

```
<authentication>
  <truststore path="..." relative-to="..." keystore-password="..."/>
</authentication>
```

This element is used to define how to load a key store file that can be used as the trust store within the SSLContext we create internally, the store is then used to verify the certificates of the remote side of the connection be that inbound or outbound.

- **path** (mandatory) - This is the path to the keystore, this can be an absolute path or relative to the next attribute.
- **relative-to** (optional) - The name of a service representing a path the keystore is relative to.
- **keystore-password** (mandatory) - The password required to open the keystore.

Although this is a definition of a trust store the attribute for the password is `keystore-password`, this is because the underlying file being opened is still a key store.

<local />

```
<authentication>
  <local default-user="..." allowed-users="..." />
</authentication>
```

This element switches on the local authentication mechanism that allows clients to the server to verify that they are local to the server, at the protocol level it is optional for the remote client to send a user name in the authentication response.

- **default-user** (optional) - If the client does not pass in a username this is the assumed username, this value is also automatically added to the list of allowed-users.
- **allowed-users** (optional) - This attribute is used to specify a comma separated list of users allowed to authenticate using the local mechanism, alternatively '*' can be specified to allow any username to be specified.

<jaas />

```
<authentication>
  <jaas name="..." />
</authentication>
```

The jaas element is used to enable username and password based authentication where the supplied username and password are verified by making use of a configured jaas domain.

- **name** (mandatory) - The name of the jaas domain to use to verify the supplied username and password.

As JAAS authentication works by taking a username and password and verifying these the use of this element means that at the transport level authentication will be forced to send the password in plain text, any interception of the messages exchanged between the client and server without SSL enabled will reveal the users password.

<ldap />

```
<authentication>
  <ldap connection="..." base-dn="..." recursive="..." user-dn="...">
    <username-filter attribute="..." />
    <advanced-filter filter="..." />
  </ldap>
</authentication>
```

The ldap element is used to define how LDAP searches will be used to authenticate a user, this works by first connecting to LDAP and performing a search using the supplied user name to identity the distinguished name of the user and then a subsequent connection is made to the server using the password supplied by the user - if this second connection is a success then authentication succeeds.

Due to the verification approach used this configuration causes the authentication mechanisms selected for the protocol to cause the password to be sent from the client in plain text, the following Jira issue is to investigating proxying a Digest authentication with the LDAP server so no plain text password is needed AS7-4195.

- **connection** (mandatory) - The name of the connection to use to connect to LDAP.
- **base-dn** (mandatory) - The distinguished name of the context to use to begin the search from.
- **recursive** (optional) - Should the filter be executed recursively? Defaults to false.
- **user-dn** (optional) - After the user has been found specifies which attribute to read for the users distinguished name, defaults to 'dn'.

Within the ldap element only one of `<username-filter />` or `<advanced-filter />` can be specified.

<username-filter />

This element is used for a simple filter to match the username specified by the remote user against a single attribute, as an example with Active Directory the match is most likely to be against the 'sAMAccountName' attribute.

- **attribute** (mandatory) - The name of the field to match the users supplied username against.

<advanced-filter />

This element is used where a more advanced filter is required, one example use of this filter is to exclude certain matches by specifying some additional criteria for the filter.

- **filter** (mandatory) - The filter to execute to locate the user, this filter should contain '{0}' as a place holder for the username supplied by the user authenticating.

<properties />

```
<authentication>
  <properties path="..." relative-to="..." plain-text="..." />
</authentication>
```

The properties element is used to reference a properties file to load to read a users password or pre-prepared digest for the authentication process.

- **path** (mandatory) - The path to the properties file, either absolute or relative to the path referenced by the relative-to attribute.
- **relative-to** (optional) - The name of a path service that the defined path will be relative to.
- **plain-text** (optional) - Setting to specify if the passwords are stored as plain text within the properties file, defaults to false.

By default the properties files are expected to store a pre-prepared hash of the users password in the form HEX(MD5(username ':' realm ':' password))

<users />

```
<authentication>
  <users>
    <user username="...">
      <password>...</password>
    </user>
  </users>
</authentication>
```

This is a very simple store of a username and password that stores both of these within the domain model, this is only really provided for the provision of simple examples.

- **username** (mandatory) - A users username.

The `<password/>` element is then used to define the password for the user.

<authorization />

The authorization element is used to define how a users roles can be loaded after the authentication process completes, these roles may then be used for subsequent authorization decisions based on the service being accessed. At the moment only a properties file approach or a custom plug-in are supported - support for loading roles from LDAP or from a database are planned for a subsequent release.

```
<authorization>
  <properties />
  <plug-in />
</authorization>
```

<properties />

```
<authorization>
  <properties path="..." relative-to="..." />
</authorization>
```

The format of the properties file is `username={ROLES}` where {ROLES} is a comma separated list of the users roles.

- **path** (mandatory) - The path to the properties file, either absolute or relative to the path referenced by the relative-to attribute.
- **relative-to** (optional) - The name of a path service that the defined path will be relative to.

<outbound-connection />

Strictly speaking these are not a part of the security realm definition, however at the moment they are only used by security realms so the definition of outbound connection is described here.

```
<management>
  <security-realms />
  <outbound-connections>
    <ldap />
  </outbound-connections>
</management>
```

<ldap />

At the moment we only support outbound connections to ldap servers for the authentication process - this will later be expanded when we add support for database based authentication.

```
<outbound-connections>
  <ldap name="..." url="..." search-dn="..." search-credential="..."
initial-context-factory="..." />
</outbound-connections>
```

The outbound connections are defined in this section and then referenced by name from the configuration that makes use of them.

- **name** (mandatory) - The unique name used to reference this connection.
- **url** (mandatory) - The URL use to establish the LDAP connection.
- **search-dn** (mandatory) - The distinguished name of the user to authenticate as to perform the searches.
- **search-credential** (mandatory) - The password required to connect to LDAP as the search-dn.
- **initial-context-factory** (optional) - Allows overriding the initial context factory, defaults to '`com.sun.jndi.ldap.LdapCtxFactory`'

5.21.13 Examples

This section of the document contains a couple of examples for the most common scenarios likely to be used with the security realms, please feel free to raise Jira issues requesting additional scenarios or if you have configured something not covered here please feel free to add your own examples - this document is editable after all 

At the moment these examples are making use of the '`ManagementRealm`' however the same can apply to the '`ApplicationRealm`' or any custom realm you create for yourselves.

LDAP Authentication

The following example demonstrates an example configuration making use of Active Directory to verify the users username and password.

```
<management>
  <security-realms>
    <security-realm name="ManagementRealm">
      <authentication>
        <ldap connection="EC2" base-dn="CN=Users,DC=darranl,DC=jboss,DC=org">
          <username-filter attribute="sAMAccountName" />
        </ldap>
      </authentication>
    </security-realm>

  </security-realms>

  <outbound-connections>
    <ldap name="EC2" url="ldap://127.0.0.1:9797"
search-dn="CN=wf8,CN=Users,DC=darranl,DC=jboss,DC=org" search-credential="password"/>
  </outbound-connections>

  ...

</management>
```

For simplicity the `<local/>` configuration has been removed from this example, however there it is fine to leave that in place for local authentication to remain possible.

Enable SSL

The first step is the creation of the key, by default this is going to be used for both the native management interface and the http management interface - to create the key we can use the `keyTool`, the following example will create a key valid for one year.

Open a terminal window in the folder `{jboss.home}/standalone/configuration` and enter the following command: -

```
keytool -genkey -alias server -keyalg RSA -keystore server.keystore -validity 365
```

```
Enter keystore password:
Re-enter new password:
```

In this example I choose '`keystore_password`'.

```
What is your first and last name?
  [Unknown]:  localhost
```

Of all of the questions asked this is the most important and should match the host name that will be entered into the web browser to connect to the admin console.

Answer the remaining questions as you see fit and at the end for the purpose of this example I set the key password to '`key_password`'.

The following example shows how this newly created keystore will be referenced to enable SSL.

```
<security-realm name="ManagementRealm">
  <server-identities>
    <ssl>
      <keystore path="server.keystore" relative-to="jboss.server.config.dir"
keystore-password="keystore_password" alias="server" key-password="key_password" />
    </ssl>
  </server-identities>
  <authentication>
    ...
  </authentication>
</security-realm>
```

The contents of the `<authentication />` have not been changed in this example so authentication still occurs using either the local mechanism or username/password authentication using Digest.

Add Client-Cert to SSL

To enable Client-Cert style authentication we just now need to add a `<truststore />` element to the `<authentication />` element referencing a trust store that has had the certificates or trusted clients imported.

```
<security-realm name="ManagementRealm">
  <server-identities>
    <ssl>
      <keystore path="server.keystore" relative-to="jboss.server.config.dir"
keystore-password="keystore_password" alias="server" key-password="key_password" />
    </ssl>
  </server-identities>
  <authentication>
    <truststore path="server.truststore" relative-to="jboss.server.config.dir"
keystore-password="truststore_password" />
    <local default-user="$local"/>
    <properties path="mgmt-users.properties" relative-to="jboss.server.config.dir"/>
  </authentication>
</security-realm>
```

In this scenario if Client-Cert authentication does not occur clients can fall back to use either the local mechanism or username/password authentication. To make Client-Cert based authentication mandatory just remove the `<local />` and `<properties />` elements.

5.21.14 Plug Ins

Within WildFly 8 for communication with the management interfaces and for other services exposed using Remoting where username / password authentication is used the use of Digest authentication is preferred over the use of HTTP Basic or SASL Plain so that we can avoid the sending of password in the clear over the network. For validation of the digests to work on the server we either need to be able to retrieve a users plain text password or we need to be able to obtain a ready prepared hash of their password along with the username and realm.

Previously to allow the addition of custom user stores we have added an option to the realms to call out to a JAAS domain to validate a users username and password, the problem with this approach is that to call JAAS we need the remote user to send in their plain text username and password so that a JAAS LoginModule can perform the validation, this forces us down to use either the HTTP Basic authentication mechanism or the SASL Plain mechanism depending on the transport used which is undesirable as we can not longer use Digest.

To overcome this we now support plugging in custom user stores to support loading a users password, hash and roles from a custom store to allow different stores to be implemented without forcing the authentication back to plain text variant, this article describes the requirements for a plug in and shows a simple example plug-in for use with WildFly 8.

When implementing a plug in there are two steps to the authentication process, the first step is to load the users identity and credential from the relevant store - this is then used to verify the user attempting to connect is valid. After the remote user is validated we then load the users roles in a second step. For this reason the support for plug-ins is split into the two stages, when providing a plug-in either of these two steps can be implemented but there is no requirement to implement the other side.

When implementing a plug-in the following interfaces are the bare minimum that need to be implemented so depending on if a plug-in to load a users identity or a plug-in to load a users roles is being implemented you will be implementing one of these interfaces.

***Note** - All classes and interfaces of the SPI to be implemented are in the 'org.jboss.as.domain.management.plugin' package which is a part of the 'org.jboss.as.domain-management' module but for simplicity for the rest of this section only the short names will be shown.*

AuthenticationPlugIn

To implement an `AuthenticationPlugIn` the following interface needs to be implemened: -

```
public interface AuthenticationPlugIn<T extends Credential> {
    Identity<T> loadIdentity(final String userName, final String realm) throws IOException;
}
```

During the authentication process this method will be called with the user name supplied by the remote user and the name of the realm they are authenticating against, this method call represents that an authentication attempt is occurring but it is the Identity instance that is returned that will be used for the actual authentication to verify the remote user.

The Identity interface is also an interface you will implement: -

```
public interface Identity<T extends Credential> {
    String getUserName();
    T getCredential();
}
```

Additional information can be contained within the Identity implementation although it will not currently be used, the key piece of information here is the Credential that will be returned - this needs to be one of the following: -

PasswordCredential

```
public final class PasswordCredential implements Credential {
    public PasswordCredential(final char[] password);
    public char[] getPassword();
    void clear();
}
```

The `PasswordCredential` is already implemented so use this class if you have the plain text password of the remote user, by using this the secured interfaces will be able to continue using the Digest mechanism for authentication.

DigestCredential

```
public final class DigestCredential implements Credential {
    public DigestCredential(final String hash);
    public String getHash();
}
```

This class is also already implemented and should be returned if instead of the plain text password you already have a pre-prepared hash of the username, realm and password.

ValidatePasswordCredential

```
public interface ValidatePasswordCredential extends Credential {
    boolean validatePassword(final char[] password);
}
```

This is a special Credential type to use when it is not possible to obtain either a plain text representation of the password or a pre-prepared hash - this is an interface as you will need to provide an implementation to verify a supplied password. The down side of using this type of Credential is that the authentication mechanism used at the transport level will need to drop down from Digest to either HTTP Basic or SASL Plain which will now mean that the remote client is sending their credential across the network in the clear.

If you use this type of credential be sure to force the mechanism choice to Plain as described in the configuration section below.

AuthorizationPlugIn

If you are implementing a custom mechanism to load a users roles you need to implement the `AuthorizationPlugIn`

```
public interface AuthorizationPlugIn {
    String[] loadRoles(final String userName, final String realm) throws IOException;
}
```

As with the `AuthenticationPlugIn` this has a single method that takes a users userName and realm - the return type is an array of Strings with each entry representing a role the user is a member of.

PlugInConfigurationSupport

In addition to the specific interfaces above there is an additional interface that a plug-in can implement to receive configuration information before the plug-in is used and also to receive a Map instance that can be used to share state between the plug-in instance used for the authentication step of the call and the plug-in instance used for the authorization step.

```
public interface PlugInConfigurationSupport {
    void init(final Map<String, String> configuration, final Map<String, Object> sharedState)
throws IOException;
}
```

Installing and Configuring a Plug-In

The next step of this article describes the steps to implement a plug-in provider and how to make it available within WildFly 8 and how to configure it. Example configuration and an example implementation are shown to illustrate this.

The following is an example security realm definition which will be used to illustrate this: -

```
<security-realm name="PlugInRealm">
   <plug-ins>
      <plug-in module="org.jboss.as.sample.plugin"/>
   </plug-ins>
   <authentication>
      <plug-in name="Sample">
         <properties>
            <property name="darranl.password" value="dpd"/>
            <property name="darranl.roles" value="Admin,Banker,User"/>
         </properties>
      </plug-in>
   </authentication>
   <authorization>
      <plug-in name="Delegate" />
   </authorization>
</security-realm>
```

Before looking closely at the packaging and configuration there is one more interface to implement and that is the `PlugInProvider` interface, that interface is responsible for making PlugIn instances available at runtime to handle the requests.

PlugInProvider

```
public interface PlugInProvider {
    AuthenticationPlugIn<Credential> loadAuthenticationPlugIn(final String name);
    AuthorizationPlugIn loadAuthorizationPlugIn(final String name);
}
```

These methods are called with the name that is supplied in the plug-in elements that are contained within the authentication and authorization elements of the configuration, based on the sample configuration above the loadAuthenticationPlugIn method will be called with a parameter of 'Sample' and the loadAuthorizationPlugIn method will be called with a parameter of 'Delegate'.

Multiple plug-in providers may be available to the application server so if a `PlugInProvider` implementation does not recognise a name then it should just return null and the server will continue searching the other providers. If a `PlugInProvider` does recognise a name but fails to instantiate the PlugIn then a `RuntimeException` can be thrown to indicate the failure.

As a server could have many providers registered it is recommended that a naming convention including some form of hierarchy is used e.g. use package style names to avoid conflicts.

For the example the implementation is as follows: -

```
public class SamplePluginProvider implements PlugInProvider {

    public AuthenticationPlugIn<Credential> loadAuthenticationPlugIn(String name) {
        if ("Sample".equals(name)) {
            return new SampleAuthenticationPlugIn();
        }
        return null;
    }

    public AuthorizationPlugIn loadAuthorizationPlugIn(String name) {
        if ("Sample".equals(name)) {
            return new SampleAuthenticationPlugIn();
        } else if ("Delegate".equals(name)) {
            return new DelegateAuthorizationPlugIn();
        }
        return null;
    }
}
```

The load methods are called for each authentication attempt but it will be an implementation detail of the provider if it decides to return a new instance of the provider each time - in this scenario as we also use configuration and shared state then new instances of the implementations make sense.

To load the provider use a ServiceLoader so within the META-INF/services folder of the jar this project adds a file called '`org.jboss.as.domain.management.plugin.PlugInProvider`' - this contains a single entry which is the fully qualified class name of the PlugInProvider implementation class.

```
org.jboss.as.sample.SamplePluginProvider
```

Package as a Module

To make the `PlugInProvider` available to the application it is bundled as a module and added to the modules already shipped with WildFly 8.

To add as a module we first need a `module.xml`: -

```
<?xml version="1.0" encoding="UTF-8"?>

<module xmlns="urn:jboss:module:1.1" name="org.jboss.as.sample.plugin">
    <properties>
    </properties>

    <resources>
        <resource-root path="SamplePlugIn.jar"/>
    </resources>

    <dependencies>
        <module name="org.jboss.as.domain-management" />
    </dependencies>
</module>
```

The interfaces being implemented are in the '`org.jboss.as.domain-management`' module so a dependency on that module is defined, this `module.xml` is then placed in the '{ `jboss.home}/modules/org/jboss/as/sample/plugin/main`'.

The compiled classed and `META-INF/services` as described above are assembled into a jar called `SamplePlugIn.jar` and also placed into this folder.

Looking back at the sample configuration at the top of the realm definition the following element was added: -

```
<plug-ins>
      <plug-in module="org.jboss.as.sample.plugin"/>
   </plug-ins>
```

This element is used to list the modules that should be searched for plug-ins. As plug-ins are loaded during the server start up this search is a lazy search so don't expect a definition to a non existant module or to a module that does not contain a plug-in to report an error.

The AuthenticationPlugIn

The example `AuthenticationPlugIn` is implemented as: -

```
public class SampleAuthenticationPlugIn extends AbstractPlugIn {

    private static final String PASSWORD_SUFFIX = ".password";
    private static final String ROLES_SUFFIX = ".roles";
    private Map<String, String> configuration;

    public void init(Map<String, String> configuration, Map<String, Object> sharedState) throws
IOException {
        this.configuration = configuration;
        // This will allow an AuthorizationPlugIn to delegate back to this instance.
        sharedState.put(AuthorizationPlugIn.class.getName(), this);
    }

    public Identity loadIdentity(String userName, String realm) throws IOException {
        String passwordKey = userName + PASSWORD_SUFFIX;
        if (configuration.containsKey(passwordKey)) {
            return new SampleIdentity(userName, configuration.get(passwordKey));
        }
        throw new IOException("Identity not found.");
    }

    public String[] loadRoles(String userName, String realm) throws IOException {
        String rolesKey = userName + ROLES_SUFFIX;
        if (configuration.containsKey(rolesKey)) {
            String roles = configuration.get(rolesKey);
            return roles.split(",");
        } else {
            return new String[0];
        }
    }

    private static class SampleIdentity implements Identity {
        private final String userName;
        private final Credential credential;

        private SampleIdentity(final String userName, final String password) {
            this.userName = userName;
            this.credential = new PasswordCredential(password.toCharArray());
        }

        public String getUserName() {
            return userName;
        }

        public Credential getCredential() {
            return credential;
        }
    }
}
```

As you can see from this implementation there is also an additional class being extended `AbstractPlugIn` - that is simply an abstract class that implements the `AuthenticationPlugIn`, `AuthorizationPlugIn`, and `PlugInConfigurationSupport` interfaces already. The properties that were defined in the configuration are passed in as a Map and importantly for this sample the plug-in adds itself to the shared state map.

The AuthorizationPlugIn

The example implementation of the authentication plug in is as follows: -

```
public class DelegateAuthorizationPlugIn extends AbstractPlugIn {

    private AuthorizationPlugIn authorizationPlugIn;

    public void init(Map<String, String> configuration, Map<String, Object> sharedState) throws
IOException {
        authorizationPlugIn = (AuthorizationPlugIn)
sharedState.get(AuthorizationPlugIn.class.getName());
    }

    public String[] loadRoles(String userName, String realm) throws IOException {
        return authorizationPlugIn.loadRoles(userName, realm);
    }

}
```

This plug-in illustrates how two plug-ins can work together, by the `AuthenticationPlugIn` placing itself in the shared state map it is possible for the authorization plug-in to make use of it for the loadRoles implementation.

Another option to consider to achieve similar behaviour could be to provide an Identity implementation that also contains the roles and place this in the shared state map - the `AuthorizationPlugIn` can retrieve this and return the roles.

Forcing Plain Text Authentication

As mentioned earlier in this article if the `ValidatePasswordCredential` is going to be used then the authentication used at the transport level needs to be forced from Digest authentication to plain text authentication, this can be achieved by adding a mechanism attribute to the plug-in definition within the authentication element i.e.

```
<authentication>
    <plug-in name="Sample" mechanism="PLAIN">
```

5.22 Subsystem configuration

The following chapters will focus on the high level management use cases that are available through the CLI and the web interface. For a detailed description of each subsystem configuration property, please consult the respective component reference.

Schema Location

The configuration schemas can found in `$JBOSS_HOME/docs/schema`.

5.22.1 EE Subsystem Configuration

Overview

The EE subsystem provides common functionality in the Java EE platform, such as the EE Concurrency Utilities (JSR 236) and `@Resource` injection. The subsystem is also responsible for managing the lifecycle of Java EE application's deployments, that is, `.ear` files.

The EE subsystem configuration may be used to:

- customise the deployment of Java EE applications
- create EE Concurrency Utilities instances
- define the default bindings

The subsystem name is ee and this document covers EE subsystem version `2.0`, which XML namespace within WildFly XML configurations is `urn:jboss:domain:ee:2.0`. The path for the subsystem's XML schema, within WildFly's distribution, is `docs/schema/jboss-as-ee_2_0.xsd`.

Subsystem XML configuration example with all elements and attributes specified:

```
<subsystem xmlns="urn:jboss:domain:ee:2.0" >
    <global-modules>
        <module name="org.jboss.logging"
                slot="main"/>
        <module name="org.apache.log4j"
                annotations="true"
                meta-inf="true"
                services="false" />
    </global-modules>
    <ear-subdeployments-isolated>true</ear-subdeployments-isolated>
    <spec-descriptor-property-replacement>false</spec-descriptor-property-replacement>
    <jboss-descriptor-property-replacement>false</jboss-descriptor-property-replacement>
    <annotation-property-replacement>false</annotation-property-replacement>
    <concurrent>
        <context-services>
            <context-service
```

```
                    name="default"
                    jndi-name="java:jboss/ee/concurrency/context/default"
                    use-transaction-setup-provider="true" />
        </context-services>
        <managed-thread-factories>
            <managed-thread-factory
                    name="default"
                    jndi-name="java:jboss/ee/concurrency/factory/default"
                    context-service="default"
                    priority="1" />
        </managed-thread-factories>
        <managed-executor-services>
            <managed-executor-service
                    name="default"
                    jndi-name="java:jboss/ee/concurrency/executor/default"
                    context-service="default"
                    thread-factory="default"
                    hung-task-threshold="60000"
                    core-threads="5"
                    max-threads="25"
                    keepalive-time="5000"
                    queue-length="1000000"
                    reject-policy="RETRY_ABORT" />
        </managed-executor-services>
        <managed-scheduled-executor-services>
            <managed-scheduled-executor-service
                    name="default"
                    jndi-name="java:jboss/ee/concurrency/scheduler/default"
                    context-service="default"
                    thread-factory="default"
                    hung-task-threshold="60000"
                    core-threads="5"
                    keepalive-time="5000"
                    reject-policy="RETRY_ABORT" />
        </managed-scheduled-executor-services>
    </concurrent>
    <default-bindings
            context-service="java:jboss/ee/concurrency/context/default"
            datasource="java:jboss/datasources/ExampleDS"
            jms-connection-factory="java:jboss/DefaultJMSConnectionFactory"
            managed-executor-service="java:jboss/ee/concurrency/executor/default"
            managed-scheduled-executor-service="java:jboss/ee/concurrency/scheduler/default"
            managed-thread-factory="java:jboss/ee/concurrency/factory/default" />
</subsystem>
```

Java EE Application Deployment

The EE subsystem configuration allows the customisation of the deployment behaviour for Java EE Applications.

Global Modules

Global modules is a set of JBoss Modules that will be added as dependencies to the JBoss Module of every Java EE deployment. Such dependencies allows Java EE deployments to see the classes exported by the global modules.

Each global module is defined through the `module` resource, an example of its XML configuration:

```
<global-modules>
    <module name="org.jboss.logging" slot="main"/>
    <module name="org.apache.log4j" annotations="true" meta-inf="true" services="false" />
  </global-modules>
```

The only mandatory attribute is the JBoss Module `name`, the `slot` attribute defaults to `main`, and both define the JBoss Module ID to reference.

The optional `annotations` attribute, which defaults to `false`, indicates if a pre-computed annotation index should be imported from META-INF/jandex.idx

The optional `services` attribute indicates if any services exposed in META-INF/services should be made available to the deployments class loader, and defaults to `false`.

The optional `meta-inf` attribute, which defaults to `true`, indicates if the Module's `META-INF` path should be available to the deployment's class loader.

EAR Subdeployments Isolation

A flag indicating whether each of the subdeployments within a `.ear` can access classes belonging to another subdeployment within the same `.ear`. The default value is `false`, which allows the subdeployments to see classes belonging to other subdeployments within the `.ear`.

```
<ear-subdeployments-isolated>true</ear-subdeployments-isolated>
```

For example:

```
myapp.ear
|
|--- web.war
|
|--- ejb1.jar
|
|--- ejb2.jar
```

If the `ear-subdeployments-isolated` is set to false, then the classes in `web.war` can access classes belonging to `ejb1.jar` and `ejb2.jar`. Similarly, classes from `ejb1.jar` can access classes from `ejb2.jar` (and vice-versa).

This flag has no effect on the isolated classloader of the `.war` file(s), i.e. irrespective of whether this flag is set to `true` or `false`, the `.war` within a `.ear` will have a isolated classloader, and other subdeployments within that `.ear` will not be able to access classes from that `.war`. This is as per spec.

Property Replacement

The EE subsystem configuration includes flags to configure whether system property replacement will be done on XML descriptors and Java Annotations included in Java EE deployments.

System properties etc are resolved in the security context of the application server itself, not the deployment that contains the file. This means that if you are running with a security manager and enable this property, a deployment can potentially access system properties or environment entries that the security manager would have otherwise prevented.

Spec Descriptor Property Replacement

Flag indicating whether system property replacement will be performed on standard Java EE XML descriptors. If not configured this defaults to `true`, however it is set to `false` in the standard configuration files shipped with WildFly.

```
<spec-descriptor-property-replacement>false</spec-descriptor-property-replacement>
```

JBoss Descriptor Property Replacement

Flag indicating whether system property replacement will be performed on WildFly proprietary XML descriptors, such as `jboss-app.xml`. This defaults to `true`.

```
<jboss-descriptor-property-replacement>false</jboss-descriptor-property-replacement>
```

Annotation Property Replacement

Flag indicating whether system property replacement will be performed on Java annotations. The default value is `false`.

```
<annotation-property-replacement>false</annotation-property-replacement>
```

EE Concurrency Utilities

EE Concurrency Utilities (JSR 236) were introduced with Java EE 7, to ease the task of writing multithreaded Java EE applications. Instances of these utilities are managed by WildFly, and the related configuration provided by the EE subsystem.

Context Services

The Context Service is a concurrency utility which creates contextual proxies from existent objects. WildFly Context Services are also used to propagate the context from a Java EE application invocation thread, to the threads internally used by the other EE Concurrency Utilities. Context Service instances may be created using the subsystem XML configuration:

```
<context-services>
    <context-service
 name="default"
 jndi-name="java:jboss/ee/concurrency/context/default"
 use-transaction-setup-provider="true" />
  </context-services>
```

The `name` attribute is mandatory, and it's value should be a unique name within all Context Services.

The `jndi-name` attribute is also mandatory, and defines where in the JNDI the Context Service should be placed.

The optional `use-trasaction-setup-provider` attribute indicates if the contextual proxies built by the Context Service should suspend transactions in context, when invoking the proxy objects, and its value defaults to true.

Management clients, such as the WildFly CLI, may also be used to configure Context Service instances. An example to `add` and `remove` one named `other`:

```
/subsystem=ee/context-service=other:add(jndi-name=java\:jboss\/ee\/concurrency\/other)
/subsystem=ee/context-service=other:remove
```

Managed Thread Factories

The Managed Thread Factory allows Java EE applications to create new threads. WildFly Managed Thread Factory instances may also, optionally, use a Context Service instance to propagate the Java EE application thread's context to the new threads. Instance creation is done through the EE subsystem, by editing the subsystem XML configuration:

```
<managed-thread-factories>
    <managed-thread-factory
 name="default"
 jndi-name="java:jboss/ee/concurrency/factory/default"
 context-service="default"
 priority="1" />
  </managed-thread-factories>
```

The `name` attribute is mandatory, and it's value should be a unique name within all Managed Thread Factories.

The `jndi-name` attribute is also mandatory, and defines where in the JNDI the Managed Thread Factory should be placed.

The optional `context-service` references an existent Context Service by its `name`. If specified then thread created by the factory will propagate the invocation context, present when creating the thread.

The optional `priority` indicates the priority for new threads created by the factory, and defaults to `5`.

Management clients, such as the WildFly CLI, may also be used to configure Managed Thread Factory instances. An example to `add` and `remove` one named `other`:

```
/subsystem=ee/managed-thread-factory=other:add(jndi-name=java\:jboss\/ee\/factory\/other)
/subsystem=ee/managed-thread-factory=other:remove
```

Managed Executor Services

The Managed Executor Service is the Java EE adaptation of Java SE Executor Service, providing to Java EE applications the functionality of asynchronous task execution. WildFly is responsible to manage the lifecycle of Managed Executor Service instances, which are specified through the EE subsystem XML configuration:

```
<managed-executor-services>
    <managed-executor-service
 name="default"
 jndi-name="java:jboss/ee/concurrency/executor/default"
 context-service="default"
 thread-factory="default"
 hung-task-threshold="60000"
 core-threads="5"
 max-threads="25"
 keepalive-time="5000"
 queue-length="1000000"
 reject-policy="RETRY_ABORT" />
  </managed-executor-services>
```

The `name` attribute is mandatory, and it's value should be a unique name within all Managed Executor Services.

The `jndi-name` attribute is also mandatory, and defines where in the JNDI the Managed Executor Service should be placed.

The optional `context-service` references an existent Context Service by its `name`. If specified then the referenced Context Service will capture the invocation context present when submitting a task to the executor, which will then be used when executing the task.

The optional `thread-factory` references an existent Managed Thread Factory by its `name`, to handle the creation of internal threads. If not specified then a Managed Thread Factory with default configuration will be created and used internally.

The mandatory `core-threads` provides the number of threads to keep in the executor's pool, even if they are idle. A value of `0` means there is no limit.

The optional `queue-length` indicates the number of tasks that can be stored in the input queue. The default value is `0`, which means the queue capacity is unlimited.

The executor’s task queue is based on the values of the attributes `core-threads` and `queue-length`:

- If `queue-length` is `0`, or `queue-length` is `Integer.MAX_VALUE (2147483647)` and `core-threads` is `0`, direct handoff queuing strategy will be used and a synchronous queue will be created.
- If `queue-length` is `Integer.MAX_VALUE` but `core-threads` is not `0`, an unbounded queue will be used.
- For any other valid value for `queue-length`, a bounded queue wil be created.

The optional `hung-task-threshold` defines a threshold value, in milliseconds, to hung a possibly blocked task. A value of `0` will never hung a task, and is the default.

The optional `long-running-tasks` is a hint to optimize the execution of long running tasks, and defaults to `false`.

The optional `max-threads` defines the the maximum number of threads used by the executor, which defaults to Integer.MAX_VALUE (2147483647).

The optional `keepalive-time` defines the time, in milliseconds, that an internal thread may be idle. The attribute default value is `60000`.

The optional reject-policy defines the policy to use when a task is rejected by the executor. The attribute value may be the default `ABORT`, which means an exception should be thrown, or `RETRY_ABORT`, which means the executor will try to submit it once more, before throwing an exception.

Management clients, such as the WildFly CLI, may also be used to configure Managed Executor Service instances. An example to `add` and `remove` one named `other`:

```
/subsystem=ee/managed-executor-service=other:add(jndi-name=java\:jboss\/ee\/executor\/other,
core-threads=2)
/subsystem=ee/managed-executor-service=other:remove
```

Managed Scheduled Executor Services

The Managed Scheduled Executor Service is the Java EE adaptation of Java SE Scheduled Executor Service, providing to Java EE applications the functionality of scheduling task execution. WildFly is responsible to manage the lifecycle of Managed Scheduled Executor Service instances, which are specified through the EE subsystem XML configuration:

```
<managed-scheduled-executor-services>
    <managed-scheduled-executor-service
 name="default"
 jndi-name="java:jboss/ee/concurrency/scheduler/default"
 context-service="default"
 thread-factory="default"
 hung-task-threshold="60000"
 core-threads="5"
 keepalive-time="5000"
 reject-policy="RETRY_ABORT" />
  </managed-scheduled-executor-services>
```

The `name` attribute is mandatory, and it's value should be a unique name within all Managed Scheduled Executor Services.

The `jndi-name` attribute is also mandatory, and defines where in the JNDI the Managed Scheduled Executor Service should be placed.

The optional `context-service` references an existent Context Service by its `name`. If specified then the referenced Context Service will capture the invocation context present when submitting a task to the executor, which will then be used when executing the task.

The optional `thread-factory` references an existent Managed Thread Factory by its `name`, to handle the creation of internal threads. If not specified then a Managed Thread Factory with default configuration will be created and used internally.

The mandatory `core-threads` provides the number of threads to keep in the executor's pool, even if they are idle. A value of `0` means there is no limit.

The optional `hung-task-threshold` defines a threshold value, in milliseconds, to hung a possibly blocked task. A value of `0` will never hung a task, and is the default.

The optional `long-running-tasks` is a hint to optimize the execution of long running tasks, and defaults to `false`.

The optional `keepalive-time` defines the time, in milliseconds, that an internal thread may be idle. The attribute default value is `60000`.

The optional reject-policy defines the policy to use when a task is rejected by the executor. The attribute value may be the default `ABORT`, which means an exception should be thrown, or`RETRY_ABORT`, which means the executor will try to submit it once more, before throwing an exception.

Management clients, such as the WildFly CLI, may also be used to configure Managed Scheduled Executor Service instances. An example to `add` and `remove` one named `other`:

```
/subsystem=ee/managed-scheduled-executor-service=other:add(jndi-name=java\:jboss\/ee\/scheduler\/o
core-threads=2)
/subsystem=ee/managed-scheduled-executor-service=other:remove
```

Default EE Bindings

The Java EE Specification mandates the existence of a default instance for each of the following resources:

- Context Service
- Datasource
- JMS Connection Factory
- Managed Executor Service
- Managed Scheduled Executor Service
- Managed Thread Factory

The EE subsystem looks up the default instances from JNDI, using the names in the default bindings configuration, before placing those in the standard JNDI names, such as `java:comp/DefaultManagedExecutorService`:

```
<default-bindings
 context-service="java:jboss/ee/concurrency/context/default"
 datasource="java:jboss/datasources/ExampleDS"
 jms-connection-factory="java:jboss/DefaultJMSConnectionFactory"
 managed-executor-service="java:jboss/ee/concurrency/executor/default"
 managed-scheduled-executor-service="java:jboss/ee/concurrency/scheduler/default"
 managed-thread-factory="java:jboss/ee/concurrency/factory/default" />
```

The default bindings are optional, if the jndi name for a default binding is not configured then the related resource will not be available to Java EE applications.

5.22.2 Naming

Overview

The Naming subsystem provides the JNDI implementation on WildFly, and its configuration allows to:

- bind entries in global JNDI namespaces
- turn off/on the remote JNDI interface

The subsystem name is naming and this document covers Naming subsystem version `2.0`, which XML namespace within WildFly XML configurations is `urn:jboss:domain:naming:2.0`. The path for the subsystem's XML schema, within WildFly's distribution, is `docs/schema/jboss-as-naming_2_0.xsd`.

Subsystem XML configuration example with all elements and attributes specified:

```
<subsystem xmlns="urn:jboss:domain:naming:2.0">
    <bindings>
        <simple name="java:global/a" value="100" type="int" />
        <simple name="java:global/jboss.org/docs/url" value="https://docs.jboss.org"
type="java.net.URL" />
        <object-factory name="java:global/foo/bar/factory" module="org.foo.bar"
class="org.foo.bar.ObjectFactory" />
        <external-context name="java:global/federation/ldap/example"
class="javax.naming.directory.InitialDirContext" cache="true">
            <environment>
                <property name="java.naming.factory.initial"
value="com.sun.jndi.ldap.LdapCtxFactory" />
                <property name="java.naming.provider.url" value="ldap://ldap.example.com:389" />
                <property name="java.naming.security.authentication" value="simple" />
                <property name="java.naming.security.principal" value="uid=admin,ou=system" />
                <property name="java.naming.security.credentials" value="secret" />
            </environment>
        </external-context>
        <lookup name="java:global/c" lookup="java:global/b" />
    </bindings>
    <remote-naming/>
</subsystem>
```

Global Bindings Configuration

The Naming subsystem configuration allows binding entries into the following global JNDI namespaces:

- `java:global`
- `java:jboss`
- `java:`

If WildFly is to be used as a Java EE application server, then it's recommended to opt for `java:global`, since it is a standard (i.e. portable) namespace.

Four different types of bindings are supported:

- Simple
- Object Factory
- External Context
- Lookup

In the subsystem's XML configuration, global bindings are configured through the `<bindings />` XML element, as an example:

```
<bindings>
    <simple name="java:global/a" value="100" type="int" />
    <object-factory name="java:global/foo/bar/factory" module="org.foo.bar"
class="org.foo.bar.ObjectFactory" />
    <external-context name="java:global/federation/ldap/example"
class="javax.naming.directory.InitialDirContext" cache="true">
        <environment>
            <property name="java.naming.factory.initial"
value="com.sun.jndi.ldap.LdapCtxFactory" />
            <property name="java.naming.provider.url" value="ldap://ldap.example.com:389" />
            <property name="java.naming.security.authentication" value="simple" />
            <property name="java.naming.security.principal" value="uid=admin,ou=system" />
            <property name="java.naming.security.credentials" value="secret" />
        </environment>
    </external-context>
    <lookup name="java:global/c" lookup="java:global/b" />
</bindings>
```

Simple Bindings

A simple binding is a primitive or java.net.URL entry, and it is defined through the `simple` XML element. An example of its XML configuration:

```
<simple name="java:global/a" value="100" type="int" />
```

The `name` attribute is mandatory and specifies the target JNDI name for the entry.

The `value` attribute is mandatory and defines the entry's value.

The optional `type` attribute, which defaults to `java.lang.String`, specifies the type of the entry's value. Besides java.lang.String, allowed types are all the primitive types and their corresponding object wrapper classes, such as int or java.lang.Integer, and java.net.URL.

Management clients, such as the WildFly CLI, may be used to configure simple bindings. An example to `add` and `remove` the one in the XML example above:

```
/subsystem=naming/binding=java\:global\/a:add(binding-type=simple, type=int, value=100)
/subsystem=naming/binding=java\:global\/a:remove
```

Object Factories

The Naming subsystem configuration allows the binding of `javax.naming.spi.ObjectFactory` entries, through the `object-factory` XML element, for instance:

```
<object-factory name="java:global/foo/bar/factory" module="org.foo.bar"
class="org.foo.bar.ObjectFactory">
    <environment>
        <property name="p1" value="v1" />
        <property name="p2" value="v2" />
    </environment>
</object-factory>
```

The `name` attribute is mandatory and specifies the target JNDI name for the entry.

The `class` attribute is mandatory and defines the object factory's Java type.

The `module` attribute is mandatory and specifies the JBoss Module ID where the object factory Java class may be loaded from.

The optional `environment` child element may be used to provide a custom environment to the object factory.

Management clients, such as the WildFly CLI, may be used to configure object factory bindings. An example to `add` and `remove` the one in the XML example above:

```
/subsystem=naming/binding=java\:global\/foo\/bar\/factory:add(binding-type=object-factory,
module=org.foo.bar, class=org.foo.bar.ObjectFactory, environment=[p1=v1, p2=v2])
/subsystem=naming/binding=java\:global\/foo\/bar\/factory:remove
```

External Context Federation

Federation of external JNDI contexts, such as a LDAP context, are achieved by adding External Context bindings to the global bindings configuration, through the `external-context` XML element. An example of its XML configuration:

```
<external-context name="java:global/federation/ldap/example"
class="javax.naming.directory.InitialDirContext" cache="true">
    <environment>
        <property name="java.naming.factory.initial" value="com.sun.jndi.ldap.LdapCtxFactory" />
        <property name="java.naming.provider.url" value="ldap://ldap.example.com:389" />
        <property name="java.naming.security.authentication" value="simple" />
        <property name="java.naming.security.principal" value="uid=admin,ou=system" />
        <property name="java.naming.security.credentials" value="secret" />
    </environment>
</external-context>
```

The `name` attribute is mandatory and specifies the target JNDI name for the entry.

The `class` attribute is mandatory and indicates the Java initial naming context type used to create the federated context. Note that such type must have a constructor with a single environment map argument.

The optional `module` attribute specifies the JBoss Module ID where any classes required by the external JNDI context may be loaded from.

The optional `cache` attribute, which value defaults to `false`, indicates if the external context instance should be cached.

The optional `environment` child element may be used to provide the custom environment needed to lookup the external context.

Management clients, such as the WildFly CLI, may be used to configure external context bindings. An example to `add` and `remove` the one in the XML example above:

```
/subsystem=naming/binding=java\:global\/federation\/ldap\/example:add(binding-type=external-contex
cache=true, class=javax.naming.directory.InitialDirContext,
environment=[java.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory,
java.naming.provider.url=ldap\:\/\/ldap.example.com\:389,
java.naming.security.authentication=simple,
java.naming.security.principal=uid\=admin\,ou\=system, java.naming.security.credentials=
secret])

/subsystem=naming/binding=java\:global\/federation\/ldap\/example:remove
```

Some JNDI providers may fail when their resources are looked up if they do not implement properly the lookup(Name) method. Their errors would look like:

```
11:31:49,047 ERROR org.jboss.resource.adapter.jms.inflow.JmsActivation (default-threads
-1)  javax.naming.InvalidNameException: Only support CompoundName names
    at com.tibco.tibjms.naming.TibjmsContext.lookup(TibjmsContext.java:504)
    at javax.naming.InitialContext.lookup(InitialContext.java:421)
```

To work around their shortcomings, the `org.jboss.as.naming.lookup.by.string` property can be specified in the external-context's environment to use instead the lookup(String) method (with a performance degradation):

```
<property name="org.jboss.as.naming.lookup.by.string" value="true"/>
```

Binding Aliases

The Naming subsystem configuration allows the binding of existent entries into additional names, i.e. aliases. Binding aliases are specified through the `lookup` XML element. An example of its XML configuration:

```
<lookup name="java:global/c" lookup="java:global/b" />
```

The `name` attribute is mandatory and specifies the target JNDI name for the entry.

The `lookup` attribute is mandatory and indicates the source JNDI name. It can chain lookups on external contexts. For example, having an external context bounded to *java:global/federation/ldap/example*, searching can be done there by setting `lookup` attribute to *java:global/federation/ldap/example/subfolder*.

Management clients, such as the WildFly CLI, may be used to configure binding aliases. An example to `add` and `remove` the one in the XML example above:

```
/subsystem=naming/binding=java\:global\/c:add(binding-type=lookup, lookup=java\:global\/b)
/subsystem=naming/binding=java\:global\/c:remove
```

Remote JNDI Configuration

The Naming subsystem configuration may be used to (de)activate the remote JNDI interface, which allows clients to lookup entries present in a remote WildFly instance.

Only entries within the `java:jboss/exported` context are accessible over remote JNDI.

In the subsystem's XML configuration, remote JNDI access bindings are configured through the `<remote-naming />` XML element:

```
<remote-naming />
```

Management clients, such as the WildFly CLI, may be used to add/remove the remote JNDI interface. An example to `add` and `remove` the one in the XML example above:

```
/subsystem=naming/service=remote-naming:add
/subsystem=naming/service=remote-naming:remove
```

5.22.3 Data sources

Datasources are configured through the *datasource* subsystem. Declaring a new datasource consists of two separate steps: You would need to provide a JDBC driver and define a datasource that references the driver you installed.

JDBC Driver Installation

The recommended way to install a JDBC driver into WildFly 8 is to deploy it as a regular JAR deployment. The reason for this is that when you run WildFly in domain mode, deployments are automatically propagated to all servers to which the deployment applies; thus distribution of the driver JAR is one less thing for you to worry about!

Any JDBC 4-compliant driver will automatically be recognized and installed into the system by name and version. A JDBC JAR is identified using the Java service provider mechanism. Such JARs will contain a text a file named `META-INF/services/java.sql.Driver`, which contains the name of the class(es) of the Drivers which exist in that JAR. If your JDBC driver JAR is not JDBC 4-compliant, it can be made deployable in one of a few ways.

Modify the JAR

The most straightforward solution is to simply modify the JAR and add the missing file. You can do this from your command shell by:

1. Change to, or create, an empty temporary directory.
2. Create a `META-INF` subdirectory.
3. Create a `META-INF/services` subdirectory.
4. Create a `META-INF/services/java.sql.Driver` file which contains one line - the fully-qualified class name of the JDBC driver.
5. Use the `jar` command-line tool to update the JAR like this:

```
jar \-uf jdbc-driver.jar META-INF/services/java.sql.Driver
```

For a detailed explanation how to deploy JDBC 4 compliant driver jar, please refer to the chapter "Application Deployment".

Datasource Definitions

The datasource itself is defined within the subsystem *datasources*:

```
<subsystem xmlns="urn:jboss:domain:datasources:4.0">
    <datasources>
        <datasource jndi-name="java:jboss/datasources/ExampleDS" pool-name="ExampleDS">
            <connection-url>jdbc:h2:mem:test;DB_CLOSE_DELAY=-1</connection-url>
            <driver>h2</driver>
            <pool>
                <min-pool-size>10</min-pool-size>
                <max-pool-size>20</max-pool-size>
                <prefill>true</prefill>
            </pool>
            <security>
                <user-name>sa</user-name>
                <password>sa</password>
            </security>
        </datasource>
        <xa-datasource jndi-name="java:jboss/datasources/ExampleXADS" pool-name="ExampleXADS">
           <driver>h2</driver>
           <xa-datasource-property name="URL">jdbc:h2:mem:test</xa-datasource-property>
           <xa-pool>
                <min-pool-size>10</min-pool-size>
                <max-pool-size>20</max-pool-size>
                <prefill>true</prefill>
           </xa-pool>
           <security>
                <user-name>sa</user-name>
                <password>sa</password>
           </security>
        </xa-datasource>
        <drivers>
            <driver name="h2" module="com.h2database.h2">
                <xa-datasource-class>org.h2.jdbcx.JdbcDataSource</xa-datasource-class>
            </driver>
        </drivers>
  </datasources>

</subsystem>
```

(See `standalone/configuration/standalone.xml`)

As you can see the datasource references a driver by it's logical name.

You can easily query the same information through the CLI:

```
[standalone@localhost:9990 /] /subsystem=datasources:read-resource(recursive=true)
{
    "outcome" => "success",
    "result" => {
        "data-source" => {"H2DS" => {
            "connection-url" => "jdbc:h2:mem:test;DB_CLOSE_DELAY=-1",
            "jndi-name" => "java:/H2DS",
            "driver-name" => "h2",
            "pool-name" => "H2DS",
            "use-java-context" => true,
            "enabled" => true,
            "jta" => true,
            "pool-prefill" => true,
            "pool-use-strict-min" => false,
            "user-name" => "sa",
            "password" => "sa",
            "flush-strategy" => "FailingConnectionOnly",
            "background-validation" => false,
            "use-fast-fail" => false,
            "validate-on-match" => false,
            "use-ccm" => true
        }},
        "xa-data-source" => undefined,
        "jdbc-driver" => {"h2" => {
            "driver-name" => "h2",
            "driver-module-name" => "com.h2database.h2",
            "driver-xa-datasource-class-name" => "org.h2.jdbcx.JdbcDataSource"
        }}
    }
}


[standalone@localhost:9990 /] /subsystem=datasources:installed-drivers-list
{
    "outcome" => "success",
    "result" => [{
        "driver-name" => "h2",
        "deployment-name" => undefined,
        "driver-module-name" => "com.h2database.h2",
        "module-slot" => "main",
        "driver-xa-datasource-class-name" => "org.h2.jdbcx.JdbcDataSource",
        "driver-class-name" => "org.h2.Driver",
        "driver-major-version" => 1,
        "driver-minor-version" => 3,
        "jdbc-compliant" => true
    }]
}
```

Using the web console or the CLI greatly simplifies the deployment of JDBC drivers and the creation of datasources.

The CLI offers a set of commands to create and modify datasources:

```
[standalone@localhost:9990 /] data-source --help

SYNOPSIS
  data-source --help [--properties | --commands] |
              (--name=<resource_id> (--<property>=<value>)*) |
              (<command> --name=<resource_id> (--<parameter>=<value>)*)
              [--headers={<operation_header> (;<operation_header>)*}]
DESCRIPTION
  The command is used to manage resources of type /subsystem=datasources/data-source.
[...]


[standalone@localhost:9990 /] xa-data-source --help

SYNOPSIS
  xa-data-source --help [--properties | --commands] |
                 (--name=<resource_id> (--<property>=<value>)*) |
                 (<command> --name=<resource_id> (--<parameter>=<value>)*)
                 [--headers={<operation_header> (;<operation_header>)*}]

DESCRIPTION
  The command is used to manage resources of type /subsystem=datasources/xa-data-source.

RESOURCE DESCRIPTION
  A JDBC XA data-source configuration

[...]
```

Using security domains

Information can be found at https://community.jboss.org/wiki/JBossAS7SecurityDomainModel

Component Reference

The datasource subsystem is provided by the IronJacamar project. For a detailed description of the available configuration properties, please consult the project documentation.

- IronJacamar homepage: http://ironjacamar.org/
- Project Documentation: http://ironjacamar.org/documentation.html
- Schema description: http://www.ironjacamar.org/doc/userguide/1.1/en-US/html_single/index.html#deployingds_descriptor

5.22.4 Logging

Overview

The overall server logging configuration is represented by the logging subsystem. It consists of four notable parts: `handler` configurations, `logger`, the `root logger` declarations (aka log categories) and logging profiles. Each logger does reference a handler (or set of handlers). Each handler declares the log format and output:

```
<subsystem xmlns="urn:jboss:domain:logging:3.0">
   <console-handler name="CONSOLE" autoflush="true">
       <level name="DEBUG"/>
       <formatter>
           <named-formatter name="COLOR-PATTERN"/>
       </formatter>
   </console-handler>
   <periodic-rotating-file-handler name="FILE" autoflush="true">
       <formatter>
           <named-formatter name="PATTERN"/>
       </formatter>
       <file relative-to="jboss.server.log.dir" path="server.log"/>
       <suffix value=".yyyy-MM-dd"/>
   </periodic-rotating-file-handler>
   <logger category="com.arjuna">
       <level name="WARN"/>
   </logger>
   [...]
   <root-logger>
       <level name="DEBUG"/>
       <handlers>
           <handler name="CONSOLE"/>
           <handler name="FILE"/>
       </handlers>
   </root-logger>
   <formatter name="PATTERN">
       <pattern-formatter pattern="%d{yyyy-MM-dd HH:mm:ss,SSS} %-5p [%c] (%t) %s%e%n"/>
   </formatter>
   <formatter name="COLOR-PATTERN">
       <pattern-formatter pattern="%K{level}%d{HH:mm:ss,SSS} %-5p [%c] (%t) %s%e%n"/>
   </formatter>
</subsystem>
```

Attributes

The root resource contains two notable attributes `add-logging-api-dependencies` and `use-deployment-logging-config`.

logging-api-dependencies

The `add-logging-api-dependencies` controls whether or not the container adds implicit logging API dependencies to your deployments. If set to `true`, the default, all the implicit logging API dependencies are added. If set to `false` the dependencies are not added to your deployments.

deployment-logging-config

The `use-deployment-logging-config` controls whether or not your deployment is scanned for per-deployment logging. If set to `true`, the default, per-deployment logging is enabled. Set to `false` to disable this feature.

deployment Logging

Per-deployment logging allows you to add a logging configuration file to your deployment and have the logging for that deployment configured according to the configuration file. In an EAR the configuration should be in the `META-INF` directory. In a WAR or JAR deployment the configuration file can be in either the `META-INF` or `WEB-INF/classes` directories.

The following configuration files are allowed:

- logging.properties
- jboss-logging.properties
- log4j.properties
- log4j.xml
- jboss-log4j.xml

You can also disable this functionality by changing the `use-deployment-logging-config` attribute to `false`.

Logging Profiles

Logging profiles are like additional logging subsystems. Each logging profile constists of three of the four notable parts listed above: `handler` configurations, `logger` and the `root logger` declarations.

You can assign a logging profile to a deployment via the deployments manifest. Add a `Logging-Profile` entry to the `MANIFEST.MF` file with a value of the logging profile id. For example a logging profile defined on `/subsystem=logging/logging-profile=ejbs` the MANIFEST.MF would look like:

```
Manifest-Version: 1.0
Logging-Profile: ejbs
```

A logging profile can be assigned to any number of deployments. Using a logging profile also allows for runtime changes to the configuration. This is an advantage over the per-deployment logging configuration as the redeploy is not required for logging changes to take affect.

Default Log File Locations

Managed Domain

In a managed domain two types of log files do exist: Controller and server logs. The controller components govern the domain as whole. It's their responsibility to start/stop server instances and execute managed operations throughout the domain. Server logs contain the logging information for a particular server instance. They are co-located with the host the server is running on.

For the sake of simplicity we look at the default setup for managed domain. In this case, both the domain controller components and the servers are located on the same host:

Process	Log File
Host Controller	./domain/log/host-controller.log
Process Controller	./domain/log/process-controller.log
"Server One"	./domain/servers/server-one/log/server.log
"Server Two"	./domain/servers/server-two/log/server.log
"Server Three"	./domain/servers/server-three/log/server.log

Standalone Server

The default log files for a standalone server can be found in the log subdirectory of the distribution:

Process	Log File
Server	./standalone/log/server.log

Filter Expressions

Filter Type	Expression	Description	Parameter(s)	Examples
accept	accept	Accepts all log messages.	None	accept
deny	deny	enies all log messages.	None	deny
not	not(filterExpression)	Accepts a filter as an argument and inverts the returned value.	The expression takes a single filter for it's argument.	not(match("JBAS'
all	all(filterExpressions)	A filter consisting of several filters in a chain. If any filter find the log message to be unloggable, the message will not be logged and subsequent filters will not be checked.	The expression takes a comma delimited list of filters for it's argument.	all(match("JBAS") match("WELD"))
any	any(filterExpressions)	A filter consisting of several filters in a chain. If any filter fins the log message to be loggable, the message will be logged and the subsequent filters will not be checked.	The expression takes a comma delimited list of filters for it's argument.	any(match("JBAS match("WELD"))
levelChange	levelChange(level)	A filter which modifies the log record with a new level.	The expression takes a single string based level for it's argument.	levelChange(WAF

levels	levels(levels)	A filter which includes log messages with a level that is listed in the list of levels.	The expression takes a comma delimited list of string based levels for it's argument.	levels(DEBUG, IN WARN, ERROR)
levelRange	levelRange([minLevel,maxLevel])	A filter which logs records that are within the level range.	The filter expression uses a "[" to indicate a minimum inclusive level and a "]" to indicate a maximum inclusive level. Otherwise use "(" or ")" respectively indicate exclusive. The first argument for the expression is the minimum level allowed, the second argument is the maximum level allowed.	• minimum le be less tha ERROR ar maximum l be greater DEBUG levelRange DEBUG) • minimum le be less tha equal to EF and the ma level must greater tha DEBUG levelRange DEBUG) • minimum le be less tha equal to EF and the ma level must greater or e INFO levelRange INFO]
match	match("pattern")	A regular-expression based filter. The raw unformatted message is used against the pattern.	The expression takes a regular expression for it's argument. match("JBAS\d+")	

substitute	substitute("pattern", "replacement value")	A filter which replaces the first match to the pattern with the replacement value.	The first argument for the expression is the pattern the second argument is the replacement text.	substitute("JBAS"
substituteAll	substituteAll("pattern", "replacement value")	A filter which replaces all matches of the pattern with the replacement value.	The first argument for the expression is the pattern the second argument is the replacement text.	substituteAll("JBA "EAP")

List Log Files and Reading Log Files

Log files can be listed and viewed via management operations. The log files allowed to be viewed are intentionally limited to files that exist in the `jboss.server.log.dir` and are associated with a known file handler. Known file handler types include `file-handler`, `periodic-rotating-file-handler` and `size-rotating-file-handler`. The operations are valid in both standalone and domain modes.

List Log Files

The logging subsystem has a `log-file` resource off the subsystem root resource and off each `logging-profile` resource to list each log file.

CLI command and output

```
[standalone@localhost:9990 /] /subsystem=logging:read-children-names(child-type=log-file)
{
    "outcome" => "success",
    "result" => [
        "server.log",
        "server.log.2014-02-12",
        "server.log.2014-02-13"
    ]
}
```

Read Log File

The `read-log-file` operation is available on each `log-file` resource. This operation has 4 optional parameters.

Name	Description
encoding	the encoding the file should be read in
lines	the number of lines from the file. A value of -1 indicates all lines should be read.
skip	the number of lines to skip before reading.
tail	true to read from the end of the file up or false to read top down.

CLI command and output

```
[standalone@localhost:9990 /] /subsystem=logging/log-file=server.log:read-log-file
{
    "outcome" => "success",
    "result" => [
        "2014-02-14 14:16:48,781 INFO  [org.jboss.as.server.deployment.scanner] (MSC service
thread 1-11) JBAS015012: Started FileSystemDeploymentService for directory
/home/jperkins/servers/wildfly-8.0.0.Final/standalone/deployments",
        "2014-02-14 14:16:48,782 INFO  [org.jboss.as.connector.subsystems.datasources] (MSC
service thread 1-8) JBAS010400: Bound data source [java:jboss/myDs]",
        "2014-02-14 14:16:48,782 INFO  [org.jboss.as.connector.subsystems.datasources] (MSC
service thread 1-15) JBAS010400: Bound data source [java:jboss/datasources/ExampleDS]",
        "2014-02-14 14:16:48,786 INFO  [org.jboss.as.server.deployment] (MSC service thread 1-9)
JBAS015876: Starting deployment of \"simple-servlet.war\" (runtime-name:
\"simple-servlet.war\")",
        "2014-02-14 14:16:48,978 INFO  [org.jboss.ws.common.management] (MSC service thread
1-10) JBWS022052: Starting JBoss Web Services - Stack CXF Server 4.2.3.Final",
        "2014-02-14 14:16:49,160 INFO  [org.wildfly.extension.undertow] (MSC service thread
1-16) JBAS017534: Registered web context: /simple-servlet",
        "2014-02-14 14:16:49,189 INFO  [org.jboss.as.server] (Controller Boot Thread)
JBAS018559: Deployed \"simple-servlet.war\" (runtime-name : \"simple-servlet.war\")",
        "2014-02-14 14:16:49,224 INFO  [org.jboss.as] (Controller Boot Thread) JBAS015961: Http
management interface listening on http://127.0.0.1:9990/management",
        "2014-02-14 14:16:49,224 INFO  [org.jboss.as] (Controller Boot Thread) JBAS015951: Admin
console listening on http://127.0.0.1:9990",
        "2014-02-14 14:16:49,225 INFO  [org.jboss.as] (Controller Boot Thread) JBAS015874:
WildFly 8.0.0.Final \"WildFly\" started in 1906ms - Started 258 of 312 services (90 services are
lazy, passive or on-demand)"
    ]
}
```

FAQ

Why is there a logging.properties file?

You may have noticed that there is a `logging.properties` file in the configuration directory. This is logging configuration is used when the server boots up until the logging subsystem kicks in. If the logging subsystem is not included in your configuration, then this would act as the logging configuration for the entire server.

The `logging.properties` file is overwritten at boot and with each change to the logging subsystem. Any changes made to the file are not persisted. Any changes made to the XML configuration or via management operations will be persisted to the `logging.properties` file and used on the next boot.

5.22.5 Web (Undertow)

Web subsystem was replaced in WildFly 8 with Undertow.

There are two main parts to the undertow subsystem, which are server and Servlet container configuration, as well as some ancillary items. Advanced topics like load balancing and failover are covered on the "High Availability Guide". The default configuration does is suitable for most use cases and provides reasonable performance settings.

Required extension:

```
<extension module="org.wildfly.extension.undertow" />
```

Basic subsystem configuration example:

```
<subsystem xmlns="urn:jboss:domain:undertow:1.0">
        <buffer-caches>
            <buffer-cache name="default" buffer-size="1024" buffers-per-region="1024"
max-regions="10"/>
        </buffer-caches>
        <server name="default-server">
            <http-listener name="default" socket-binding="http" />
            <host name="default-host" alias="localhost">
                <location name="/" handler="welcome-content" />
            </host>
        </server>
        <servlet-container name="default" default-buffer-cache="default"
stack-trace-on-error="local-only" >
            <jsp-config/>
            <persistent-sessions/>
        </servlet-container>
        <handlers>
            <file name="welcome-content" path="${jboss.home.dir}/welcome-content"
directory-listing="true"/>
        </handlers>
    </subsystem>
```

Dependencies on other subsystems:

IO Subsystem

Buffer cache configuration

The buffer cache is used for caching content, such as static files. Multiple buffer caches can be configured, which allows for separate servers to use different sized caches.

Buffers are allocated in regions, and are of a fixed size. If you are caching many small files then using a smaller buffer size will be better.

The total amount of space used can be calculated by multiplying the buffer size by the number of buffers per region by the maximum number of regions.

```
<buffer-caches>
    <buffer-cache name="default" buffer-size="1024" buffers-per-region="1024" max-regions="10"/>
  </buffer-caches>
```

Attribute	Description
buffer-size	The size of the buffers. Smaller buffers allow space to be utilised more effectively
buffers-per-region	The numbers of buffers per region
max-regions	The maximum number of regions. This controls the maximum amount of memory that can be used for caching

Server configuration

A server represents an instance of Undertow. Basically this consists of a set of connectors and some configured handlers.

```
<server name="default-server" default-host="default-host" servlet-container="default" >
```

Attribute	Description
default-host	the virtual host that will be used if an incoming request as no Host: header
servlet-container	the servlet container that will be used by this server, unless is is explicitly overriden by the deployment

Connector configuration

Undertow provides HTTP, HTTPS and AJP connectors, which are configured per server.

Common settings

The following settings are common to all connectors:

Attribute	Description
socket-binding	The socket binding to use. This determines the address and port the listener listens on.
worker	A reference to an XNIO worker, as defined in the IO subsystem. The worker that is in use controls the IO and blocking thread pool.
buffer-pool	A reference to a buffer pool as defined in the IO subsystem. These buffers are used internally to read and write requests. In general these should be at least 8k, unless you are in a memory constrained environment.
enabled	If the connector is enabled.
max-post-size	The maximum size of incoming post requests that is allowed.
buffer-pipelined-data	If responses to HTTP pipelined requests should be buffered, and send out in a single write. This can improve performance if HTTP pipe lining is in use and responses are small.
max-header-size	The maximum size of a HTTP header block that is allowed. Responses with to much data in their header block will have the request terminated and a bad request response send.
max-parameters	The maximum number of query or path parameters that are allowed. This limit exists to prevent hash collision based DOS attacks.
max-headers	The maximum number of headers that are allowed. This limit exists to prevent hash collision based DOS attacks.
max-cookies	The maximum number of cookies that are allowed. This limit exists to prevent hash collision based DOS attacks.
allow-encoded-slash	Set this to true if you want the server to decode percent encoded slash characters. This is probably a bad idea, as it can have security implications, due to different servers interpreting the slash differently. Only enable this if you have a legacy application that requires it.
decode-url	If the URL should be decoded. If this is not set to true then percent encoded characters in the URL will be left as is.
url-charset	The charset to decode the URL to.
always-set-keep-alive	If the 'Connection: keep-alive' header should be added to all responses, even if not required by spec.
disallowed-methods	A comma separated list of HTTP methods that are not allowed. HTTP TRACE is disabled by default.

HTTP Connector

```
<http-listener name="default" socket-binding="http"  />
```

Attribute	Description
certificate-forwarding	If this is set to true then the HTTP listener will read a client certificate from the SSL_CLIENT_CERT header. This allows client cert authentication to be used, even if the server does not have a direct SSL connection to the end user. This should only be enabled for servers behind a proxy that has been configured to always set these headers.
redirect-socket	The socket binding to redirect requests that require security too.
proxy-address-forwarding	If this is enabled then the X-Forwarded-For and X-Forwarded-Proto headers will be used to determine the peer address. This allows applications that are behind a proxy to see the real address of the client, rather than the address of the proxy.

HTTPS listener

Https listener provides secure access to the server. The most important configuration option is security realm which defines SSL secure context.

```
<https-listener name="default" socket-binding="https" security-realm="ssl-realm" />
```

Attribute	Description
security-realm	The security realm to use for the SSL configuration. See Security realm examples for how to configure it: Examples
verify-client	One of either NOT_REQUESTED, REQUESTED or REQUIRED. If client cert auth is in use this should be either REQUESTED or REQUIRED.
enabled-cipher-suites	A list of cypher suit names that are allowed.

AJP listener

```
<ajp-listener name="default" socket-binding="ajp" />
```

Host configuration

The host element corresponds to a virtual host.

Attribute	Description
name	The virtual host name
alias	A whitespace separated list of additional host names that should be matched
default-web-module	The name of a deployment that should be used to serve up requests that do not match anything.

Servlet container configuration

The servlet-container element corresponds to an instance of an Undertow Servlet container. Most servers will only need a single servlet container, however there may be cases where it makes sense to define multiple containers (in particular if you want applications to be isolated, so they cannot dispatch to each other using the RequestDispatcher. You can also use multiple Servlet containers to serve different applications from the same context path on different virtual hosts).

Attribute	Description
allow-non-standard-wrappers	The Servlet specification requires applications to only wrap the request/response using wrapper classes that extend from the ServletRequestWrapper and ServletResponseWrapper classes. If this is set to true then this restriction is relaxed.
default-buffer-cache	The buffer cache that is used to cache static resources in the default Servlet.
stack-trace-on-error	Can be either all, none, or local-only. When set to none Undertow will never display stack traces. When set to All Undertow will always display them (not recommended for production use). When set to local-only Undertow will only display them for requests from local addresses, where there are no headers to indicate that the request has been proxied. Note that this feature means that the Undertow error page will be displayed instead of the default error page specified in web.xml.
default-encoding	The default encoding to use for requests and responses.
use-listener-encoding	If this is true then the default encoding will be the same as that used by the listener that received the request.

JSP configuration

Session Cookie Configuration

This allows you to change the attributes of the session cookie.

Attribute	Description
name	The cookie name
domain	The cookie domain
comment	The cookie comment
http-only	If the cookie is HTTP only
secure	If the cookie is marked secure
max-age	The max age of the cookie

Persistent Session Configuration

Persistent sessions allow session data to be saved across redeploys and restarts. This feature is enabled by adding the persistent-sessions element to the server config. This is mostly intended to be a development time feature.

If the path is not specified then session data is stored in memory, and will only be persistent across redeploys, rather than restarts.

Attribute	Description
path	The path to the persistent sessions data
relative-to	The location that the path is relevant to

5.22.6 Messaging

The JMS server configuration is done through the *messaging-activemq* subsystem. In this chapter we are going outline the frequently used configuration options. For a more detailed explanation please consult the Artemis user guide (See "Component Reference").

Required Extension

The configuration options discussed in this section assume that the the `org.wildfly.extension.messaging-activemq` extension is present in your configuration. This extension is not included in the standard `standalone.xml` and `standalone-ha.xml` configurations included in the WildFly distribution. It is, however, included with the `standalone-full.xml` and `standalone-full-ha.xml` configurations.

You can add the extension to a configuration without it either by adding an `<extension module="org.wildfly.extension.messaging-activemq"/>` element to the xml or by using the following CLI operation:

```
[standalone@localhost:9990 /]/extension=org.wildfly.extension.messaging-activemq:add
```

Connectors

There are three kind of connectors that can be used to connect to WildFly JMS Server

- `in-vm-connector` can be used by a local client (i.e. one running in the same JVM as the server)
- `remote-connector` can be used by a remote client (and uses Netty over TCP for the communication)
- `http-connector` can be used by a remote client (and uses Undertow Web Server to upgrade from a HTTP connection)

JMS Connection Factories

There are three kinds of *basic* JMS `connection-factory` that depends on the type of connectors that is used.

There is also a `pooled-connection-factory` which is special in that it is essentially a configuration facade for *both* the inbound and outbound connectors of the the Artemis JCA Resource Adapter. An MDB can be configured to use a `pooled-connection-factory` (e.g. using `@ResourceAdapter`). In this context, the MDB leverages the *inbound connector* of the Artemis JCA RA. Other kinds of clients can look up the pooled-connection-factory in JNDI (or inject it) and use it to send messages. In this context, such a client would leverage the *outbound connector* of the Artemis JCA RA. A `pooled-connection-factory` is also special because:

- It is only available to local clients, although it can be configured to point to a remote server.
- As the name suggests, it is pooled and therefore provides superior performance to the clients which are able to use it. The pool size can be configured via the `max-pool-size` and `min-pool-size` attributes.
- It should only be used to *send* (i.e. produce) messages when looked up in JNDI or injected.
- It can be configured to use specific security credentials via the `user` and `password` attributes. This is useful if the remote server to which it is pointing is secured.
- Resources acquired from it will be automatically enlisted any on-going JTA transaction. If you want to send a message from an EJB using CMT then this is likely the connection factory you want to use so the send operation will be atomically committed along with the rest of the EJB's transaction operations.

To be clear, the *inbound connector* of the Artemis JCA RA (which is for consuming messages) is only used by MDBs and other JCA-based components. It is not available to traditional clients.

Both a `connection-factory` and a `pooled-connection-factory` reference a `connector` declaration.

A `remote-connector` is associated with a `socket-binding` which tells the client using the `connection-factory` where to connect.

- A `connection-factory` referencing a `remote-connector` is suitable to be used by a *remote* client to send messages to or receive messages from the server (assuming the connection-factory has an appropriately exported `entry`).
- A `pooled-connection-factory` looked up in JNDI or injected which is referencing a `remote-connector` is suitable to be used by a *local* client to send messages to a remote server granted the `socket-binding` references an `outbound-socket-binding` pointing to the remote server in question.
- A `pooled-connection-factory` used by an MDB which is referencing a `remote-connector` is suitable to consume messages from a remote server granted the `socket-binding` references an `outbound-socket-binding` pointing to the remote server in question.

An `in-vm-connector` is associated with a `server-id` which tells the client using the `connection-factory` where to connect (since multiple Artemis servers can run in a single JVM).

- A `connection-factory` referencing an `in-vm-connector` is suitable to be used by a *local* client to either send messages to or receive messages from a local server.
- A `pooled-connection-factory` looked up in JNDI or injected which is referencing an `in-vm-connector` is suitable to be used by a *local* client only to send messages to a local server.
- A `pooled-connection-factory` used by an MDB which is referencing an `in-vm-connector` is suitable only to consume messages from a local server.

A `http-connector` is associated with the `socket-binding` that represents the HTTP socket (by default, named `http`).

- A `connection-factory` referencing a `http-connector` is suitable to be used by a remote client to send messages to or receive messages from the server by connecting to its HTTP port before upgrading to the messaging protocol.
- A `pooled-connection-factory` referencing a `http-connector` is suitable to be used by a local client to send messages to a remote server granted the `socket-binding` references an `outbound-socket-binding` pointing to the remote server in question.
- A `pooled-connection-factory` used by an MDB which is referencing a `http-connector` is suitable only to consume messages from a remote server granted the `socket-binding` references an `outbound-socket-binding` pointing to the remote server in question.

The `entry` declaration of a `connection-factory` or a `pooled-connection-factory` specifies the JNDI name under which the factory will be exposed. Only JNDI names bound in the `"java:jboss/exported"` namespace are available to remote clients. If a `connection-factory` has an entry bound in the `"java:jboss/exported"` namespace a remote client would look-up the `connection-factory` using the text *after* `"java:jboss/exported"`. For example, the "`RemoteConnectionFactory`" is bound by default to `"java:jboss/exported/jms/RemoteConnectionFactory"` which means a remote client would look-up this `connection-factory` using "`jms/RemoteConnectionFactory`". A `pooled-connection-factory` should *not* have any `entry` bound in the "`java:jboss/exported`" namespace because a `pooled-connection-factory` is not suitable for remote clients.

Since JMS 2.0, a default JMS connection factory is accessible to EE application under the JNDI name `java:comp/DefaultJMSConnectionFactory.` WildFly messaging subsystem defines a `pooled-connection-factory` that is used to provide this default connection factory. Any parameter change on this `pooled-connection-factory` will be take into account by any EE application looking the default JMS provider under the JNDI name `java:comp/DefaultJMSConnectionFactory.`

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:1.0">
    <server name="default">
        [...]
        <http-connector name="http-connector"
                        socket-binding="http"
                        endpoint="http-acceptor" />
        <http-connector name="http-connector-throughput"
                        socket-binding="http"
                        endpoint="http-acceptor-throughput">
            <param name="batch-delay"
                   value="50"/>
        </http-connector>
        <in-vm-connector name="in-vm"
                         server-id="0"/>
      [...]
      <connection-factory name="InVmConnectionFactory"
                          connectors="in-vm"
                          entries="java:/ConnectionFactory" />
      <pooled-connection-factory name="activemq-ra"
                          transaction="xa"
                          connectors="in-vm"
                          entries="java:/JmsXA java:jboss/DefaultJMSConnectionFactory"/>
      [...]
   </server>
</subsystem>
```

(See standalone/configuration/standalone-full.xml)

JMS Queues and Topics

JMS queues and topics are sub resources of the messaging-actively subsystem. One can define either a `jms-queue` or `jms-topic`. Each destination *must* be given a `name` and contain at least one entry in its `entries` element (separated by whitespace).

Each entry refers to a JNDI name of the queue or topic. Keep in mind that any `jms-queue` or `jms-topic` which needs to be accessed by a remote client needs to have an entry in the "java:jboss/exported" namespace. As with connection factories, if a `jms-queue` or or `jms-topic` has an entry bound in the "java:jboss/exported" namespace a remote client would look it up using the text *after* `"java:jboss/exported`". For example, the following `jms-queue` "testQueue" is bound to "java:jboss/exported/jms/queue/test" which means a remote client would look-up this {{kms-queue} using "jms/queue/test". A local client could look it up using "java:jboss/exported/jms/queue/test", "java:jms/queue/test", or more simply "jms/queue/test":

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:1.0">
    <server name="default">
    [...]
    <jms-queue name="testQueue"
               entries="jms/queue/test java:jboss/exported/jms/queue/test" />
    <jms-topic name="testTopic"
               entries="jms/topic/test java:jboss/exported/jms/topic/test" />
</subsystem>
```

(See standalone/configuration/standalone-full.xml)

JMS endpoints can easily be created through the CLI:

```
[standalone@localhost:9990 /] jms-queue add --queue-address=myQueue --entries=queues/myQueue
```

```
[standalone@localhost:9990 /]
/subsystem=messaging-activemq/server=default/jms-queue=myQueue:read-resource
{
    "outcome" => "success",
    "result" => {
        "durable" => true,
        "entries" => ["queues/myQueue"],
        "selector" => undefined
    }
}
```

A number of additional commands to maintain the JMS subsystem are available as well:

```
[standalone@localhost:9990 /] jms-queue --help --commands
add
...
remove
To read the description of a specific command execute 'jms-queue command_name --help'.
```

Dead Letter & Redelivery

Some of the settings are applied against an address wild card instead of a specific messaging destination. The dead letter queue and redelivery settings belong into this group:

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:1.0">
   <server name="default">
      [...]
      <address-setting name="#"
                       dead-letter-address="jms.queue.DLQ"
                       expiry-address="jms.queue.ExpiryQueue"
                       [...] />
```

(See standalone/configuration/standalone-full.xml)

Security Settings for Artemis addresses and JMS destinations

Security constraints are matched against an address wildcard, similar to the DLQ and redelivery settings.

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:1.0">
   <server name="default">
      [...]
      <security-setting name="#">
          <role name="guest"
                send="true"
                consume="true"
                create-non-durable-queue="true"
                delete-non-durable-queue="true"/>
```

(See standalone/configuration/standalone-full.xml)

Security Domain for Users

By default, Artemis will use the "`other`" JAAS security domain. This domain is used to authenticate users making connections to Artemis and then they are authorized to perform specific functions based on their role(s) and the `security-settings` described above. This domain can be changed by using the `security-domain`, e.g.:

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:1.0">
   <server name="default">
       <security domain="mySecurityDomain" />
      [...]
```

Using the Elytron Subsystem

You can also use the elytron subsystem to secure the messaging-activemq subsystem.

To use an Elytron security domain:

1. Undefine the legacy security domain.

```
/subsystem=messaging-activemq/server=default:undefine-attribute(name=security-domain)
```

2. Set an Elytron security domain.

```
/subsystem=messaging-activemq/server=default:write-attribute(name=elytron-domain,
value=myElytronSecurityDomain)
```

You can only define either `security-domain` or `elytron-domain`, but you cannot have both defined at the same time. If neither is defined, WildFly will use the `security-domain` default value of `other`, which maps to the `other` legacy security domain.

Cluster Authentication

If the Artemis server is configured to be clustered, it will use the `cluster` 's `user` and `password` attributes to connect to other Artemis nodes in the cluster.

If you do not change the default value of <cluster-password>, Artemis will fail to authenticate with the error:

```
HQ224018: Failed to create session: HornetQExceptionerrorType=CLUSTER_SECURITY_EXCEPTION
message=HQ119099: Unable to authenticate cluster user: HORNETQ.CLUSTER.ADMIN.USER
```

To prevent this error, you must specify a value for `<cluster-password>`. It is possible to encrypt this value by following this guide.

Alternatively, you can use the system property jboss.messaging.cluster.password to specify the cluster password from the command line.

Deployment of -jms.xml files

Starting with WildFly 8, you have the ability to deploy a -jms.xml file defining JMS destinations, e.g.:

```
<?xml version="1.0" encoding="UTF-8"?>
<messaging-deployment xmlns="urn:jboss:messaging-activemq-deployment:1.0">
   <server name="default">
      <jms-destinations>
         <jms-queue name="sample">
            <entry name="jms/queue/sample"/>
            <entry name="java:jboss/exported/jms/queue/sample"/>
         </jms-queue>
      </jms-destinations>
   </server>
</messaging-deployment>
```

This feature **is primarily intended for development** as destinations deployed this way can not be managed with any of the provided management tools (e.g. console, CLI, etc).

JMS Bridge

The function of a JMS bridge is to consume messages from a source JMS destination, and send them to a target JMS destination. Typically either the source or the target destinations are on different servers.
The bridge can also be used to bridge messages from other non Artemis JMS servers, as long as they are JMS 1.1 compliant.

The JMS Bridge is provided by the Artemis project. For a detailed description of the available configuration properties, please consult the project documentation.

Modules for other messaging brokers

Source and target JMS resources (destination and connection factories) are looked up using JNDI.
If either the source or the target resources are managed by another messaging server than WildFly, the required client classes must be bundled in a module. The name of the module must then be declared when the JMS Bridge is configured.

The use of a JMS bridges with any messaging provider will require to create a module containing the jar of this provider.

Let's suppose we want to use an hypothetical messaging provider named AcmeMQ. We want to bridge messages coming from a source AcmeMQ destination to a target destination on the local WildFly messaging server. To lookup AcmeMQ resources from JNDI, 2 jars are required, acmemq-1.2.3.jar, mylogapi-0.0.1.jar (please note these jars do not exist, this is just for the example purpose). We must *not* include a JMS jar since it will be provided by a WildFly module directly.

To use these resources in a JMS bridge, we must bundle them in a WildFly module:

in JBOSS_HOME/modules, we create the layout:

```
modules/
`-- org
    `-- acmemq
        `-- main
            |-- acmemq-1.2.3.jar
            |-- mylogapi-0.0.1.jar
            `-- module.xml
```

We define the module in `module.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<module xmlns="urn:jboss:module:1.1" name="org.acmemq">
    <properties>
        <property name="jboss.api" value="private"/>
    </properties>


    <resources>
        <!-- insert resources required to connect to the source or target   -->
        <!-- messaging brokers if it not another WildFly instance           -->
        <resource-root path="acmemq-1.2.3.jar" />
        <resource-root path="mylogapi-0.0.1.jar" />
    </resources>


    <dependencies>
       <!-- add the dependencies required by JMS Bridge code               -->
       <module name="javax.api" />
       <module name="javax.jms.api" />
       <module name="javax.transaction.api"/>
       <module name="org.jboss.remote-naming"/>
       <!-- we depend on org.apache.activemq.artemis module since we will send messages to  -->
       <!-- the Artemis server embedded in the local WildFly instance       -->
       <module name="org.apache.activemq.artemis" />
    </dependencies>
</module>
```

Configuration

A JMS bridge is defined inside a `jms-bridge` section of the `messaging-activemq` subsystem in the XML configuration files.

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:1.0">
   <jms-bridge name="myBridge" module="org.acmemq">
      <source connection-factory="ConnectionFactory"
              destination="sourceQ"
              user="user1"
              password="pwd1"
              quality-of-service="AT_MOST_ONCE"
              failure-retry-interval="500"
              max-retries="1"
              max-batch-size="500"
              max-batch-time="500"
              add-messageID-in-header="true">
         <source-context>
            <property name="java.naming.factory.initial"
                      value="org.acmemq.jndi.AcmeMQInitialContextFactory"/>
            <property name="java.naming.provider.url"
                      value="tcp://127.0.0.1:9292"/>
         </source-context>
      </source>
      <target connection-factory"/jms/invmTargetCF"
              destination="/jms/targetQ" />
      </target>
   </jms-bridge>
</subsystem>
```

The `source` and `target` sections contain the name of the JMS resource (`connection-factory` and `destination`) that will be looked up in JNDI.
It optionally defines the `user` and `password` credentials. If they are set, they will be passed as arguments when creating the JMS connection from the looked up ConnectionFactory.
It is also possible to define JNDI context properties in the `source-context` and `target-context` sections. If these sections are absent, the JMS resources will be looked up in the local WildFly instance (as it is the case in the `target` section in the example above).

Management commands

A JMS Bridge can also be managed using the WildFly command line interface:

```
[standalone@localhost:9990 /] /subsystem=messaging/jms-bridge=myBridge/:add(module="org.acmemq",
\
      source-destination="sourceQ",
\
      source-connection-factory="ConnectionFactory",
\
      source-user="user1",
\
      source-password="pwd1",
\
      source-context={"java.naming.factory.initial" =>
"org.acmemq.jndi.AcmeMQInitialContextFactory", \
                     "java.naming.provider.url" => "tcp://127.0.0.1:9292"},
\
      target-destination="/jms/targetQ",
\
      target-connection-factory="/jms/invmTargetCF",
\
      quality-of-service=AT_MOST_ONCE,
\
      failure-retry-interval=500,
\
      max-retries=1,
\
      max-batch-size=500,
\
      max-batch-time=500,
\
      add-messageID-in-header=true)
{"outcome" => "success"}
```

You can also see the complete JMS Bridge resource description from the CLI:

```
[standalone@localhost:9990 /] /subsystem=messaging/jms-bridge=*/:read-resource-description
{
    "outcome" => "success",
    "result" => [{
        "address" => [
            ("subsystem" => "messaging"),
            ("jms-bridge" => "*")
        ],
        "outcome" => "success",
        "result" => {
            "description" => "A JMS bridge instance.",
            "attributes" => {
                ...
        }
    }]
}
```

Component Reference

The messaging-activemq subsystem is provided by the Artemis project. For a detailed description of the available configuration properties, please consult the project documentation.

- Artemis Homepage: http://activemq.apache.org/artemis/
- Artemis User Documentation: http://activemq.apache.org/artemis/docs.html

5.22.7 Security

The security subsystem is the subsystem that brings the security services provided by PicketBox to the WildFly 8 server instances.

If you are looking to secure the management interfaces for the management of the domain then you should read the Securing the Management Interfaces chapter as the management interfaces themselves are not run within a WildFly process so use a custom configuration.

Structure of the Security Subsystem

When deploying applications to WildFly most of the time it is likely that you would be deploying a web application or EJBs and just require a security domain to be defined with login modules to verify the users identity, this chapter aims to provide additional detail regarding the architecture and capability of the security subsystem however if you are just looking to define a security domain and leave the rest to the container please jump to the security-domains section.

The security subsystem operates by using a security context associated with the current request, this security context then makes available to the relevant container a number of capabilities from the configured security domain, the capabilities exposed are an authentication manager, an authorization manager, an audit manager and a mapping manager.

Authentication Manager

The authentication manager is the component that performs the actual authentication taking the declared users identity and their credential so that the login context for the security domain can be used to 'login' the user using the configured login module or modules.

Authorization Manager

The authorization manager is a component which can be obtained by the container from the current security context to either obtain information about a users roles or to perform an authorization check against a resource for the currently authenticated user.

Audit Manager

The audit manager from the security context is the component that can be used to log audit events in relation to the security domain.

Mapping Manager

The mapping manager can be used to assign additional principals, credentials, roles or attributes to the authenticated subject.

Security Subsystem Configuration

By default a lot of defaults have already been selected for the security subsystem and unless there is a specific implementation detail you need to change, these defaults should not require modification. This chapter describes all of the possible configuration attributes for completeness but do keep in mind that not all will need to be changed.

The security subsystem is enabled by default by the addition of the following extension: -

<extension module="org.jboss.as.security"/>

The namespace used for the configuration of the security subsystem is urn:jboss:domain:security:1.0, the configuration is defined within the <subsystem> element from this namespace.

The <subsystem> element can optionally contain the following child elements.

- security-management
- subject-factory
- security-domains
- security-properties

security-management

This element is used to override some of the high level implementation details of the PicketBox implementation if you have a need to change some of this behaviour.

The element can have any or the following attributes set, all of which are optional.

authentication-manager-class-name	Specifies the AuthenticationManager implementation class name to use.
deep-copy-subject-mode	Sets the copy mode of subjects done by the security managers to be deep copies that makes copies of the subject principals and credentials if they are cloneable. It should be set to true if subject include mutable content that can be corrupted when multiple threads have the same identity and cache flushes/logout clearing the subject in one thread results in subject references affecting other threads. Default value is "false".
default-callback-handler-class-name	Specifies a global class name for the CallbackHandler implementation to be used with login modules.
authorization-manager-class-name	Attribute specifies the AuthorizationManager implementation class name to use.
audit-manager-class-name	Specifies the AuditManager implementation class name to use.
identity-trust-manager-class-name	Specifies the IdentityTrustManager implementation class name to use.
mapping-manager-class-name	Specifies the MappingManager implementation class name to use.

subject-factory

The subject factory is responsible for creating subject instances, this also makes use of the authentication manager to actually verify the caller. It is used mainly by JCA components to establish a subject. It is not likely this would need to be overridden but if it is required the "subject-factory-class-name" attribute can be specified on the subject-factory element.

security-domains

This portion of the configuration is where the bulk of the security subsystem configuration will actually take place for most administrators, the security domains contain the configuration which is specific to a deployment.

The security-domains element can contain numerous <security-domain> definitions, a security-domain can have the following attributes set:

name	The unique name of this security domain.
~~extends~~	Although version 1.0 of the security subsystem schema contained an 'extends' attribute, security domain inheritance is not supported and this attribute should not be used.
cache-type	The type of authentication cache to use with this domain. If this attribute is removed no cache will be used. Allowed values are "default" or "infinispan"

The following elements can then be set within the security-domain to configure the domain behaviour.

authentication

The authentication element is used to hold the list of login modules that will be used for authentication when this domain is used, the structure of the login-module element is:

```
<login-module code="..." flag="..." module="...">
  <module-option name="..." value="..."/>
</login-module>
```

The code attribute is used to specify the implementing class of the login module which can either be the full class name or one of the abbreviated names from the following list:

Code	Classname
Client	`org.jboss.security.ClientLoginModule`
Certificate	`org.jboss.security.auth.spi.BaseCertLoginModule`
CertificateUsers	`org.jboss.security.auth.spi.BaseCertLoginModule`
CertificateRoles	`org.jboss.security.auth.spi.CertRolesLoginModule`
Database	`org.jboss.security.auth.spi.DatabaseServerLoginModule`
DatabaseCertificate	`org.jboss.security.auth.spi.DatabaseCertLoginModule`
DatabaseUsers	`org.jboss.security.auth.spi.DatabaseServerLoginModule`
Identity	`org.jboss.security.auth.spi.IdentityLoginModule`
Ldap	`org.jboss.security.auth.spi.LdapLoginModule`
LdapExtended	`org.jboss.security.auth.spi.LdapExtLoginModule`
RoleMapping	`org.jboss.security.auth.spi.RoleMappingLoginModule`
RunAs	`org.jboss.security.auth.spi.RunAsLoginModule`
Simple	`org.jboss.security.auth.spi.SimpleServerLoginModule`
ConfiguredIdentity	`org.picketbox.datasource.security.ConfiguredIdentityLoginModule`
SecureIdentity	`org.picketbox.datasource.security.SecureIdentityLoginModule`
PropertiesUsers	`org.jboss.security.auth.spi.PropertiesUsersLoginModule`
SimpleUsers	`org.jboss.security.auth.spi.SimpleUsersLoginModule`
LdapUsers	`org.jboss.security.auth.spi.LdapUsersLoginModule`
Kerberos	`com.sun.security.auth.module.Krb5LoginModule`
SPNEGOUsers	`org.jboss.security.negotiation.spnego.SPNEGOLoginModule`
AdvancedLdap	`org.jboss.security.negotiation.AdvancedLdapLoginModule`
AdvancedADLdap	`org.jboss.security.negotiation.AdvancedADLoginModule`
UsersRoles	`org.jboss.security.auth.spi.UsersRolesLoginModule`

The module attribute specifies the name of the JBoss Modules module from which the class specified by the code attribute should be loaded. Specifying it is not necessary if one of the abbreviated names in the above list is used.

The flag attribute is used to specify the JAAS flag for this module and should be one of required, requisite, sufficient, or optional.

The module-option element can be repeated zero or more times to specify the module options as required for the login module being configured. It requires the name and value attributes.

See Authentication Modules for further details on the various modules listed above.

authentication-jaspi

The authentication-jaspi is used to configure a Java Authentication SPI (JASPI) provider as the authentication mechanism. A security domain can have either a <authentication> or a <authentication-jaspi> element, but not both. We set up JASPI by configuring one or more login modules inside the login-module-stack element and setting up an authentication module. Here is the structure of the authentication-jaspi element:

```
<login-module-stack name="...">
  <login-module code="..." flag="..." module="...">
    <module-option name="..." value="..."/>
  </login-module>
</login-module-stack>
<auth-module code="..." login-module-stack-ref="...">
  <module-option name="..." value="..."/>
</auth-module>
```

The login-module-stack-ref attribute value must be the name of the login-module-stack element to be used. The sub-element login-module is configured just like in the authentication part

authorization

Authorization in the AS container is normally done with RBAC (role based access control) but there are situations where a more fine grained authorization policy is required. The authorization element allows definition of different authorization modules to used, such that authorization can be checked with JACC (Java Authorization Contract for Containers) or XACML (eXtensible Access Control Markup Language). The structure of the authorization element is:

```
<policy-module code="..." flag="..." module="...">
  <module-option name="..." value="..."/>
</policy-module>
```

The code attribute is used to specify the implementing class of the policy module which can either be the full class name or one of the abbreviated names from the following list:

Code	Classname
DenyAll	`org.jboss.security.authorization.modules.AllDenyAuthorizationModule`
PermitAll	`org.jboss.security.authorization.modules.AllPermitAuthorizationModule`
Delegating	`org.jboss.security.authorization.modules.DelegatingAuthorizationModule`
Web	`org.jboss.security.authorization.modules.WebAuthorizationModule`
JACC	`org.jboss.security.authorization.modules.JACCAuthorizationModule`
XACML	`org.jboss.security.authorization.modules.XACMLAuthorizationModule`

The module attribute specifies the name of the JBoss Modules module from which the class specified by the code attribute should be loaded. Specifying it is not necessary if one of the abbreviated names in the above list is used.

The flag attribute is used to specify the JAAS flag for this module and should be one of required, requisite, sufficient, or optional.

The module-option element can be repeated zero or more times to specify the module options as required for the login module being configured. It requires the name and value attributes.

mapping

The mapping element defines additional mapping of principals, credentials, roles and attributes for the subject. The structure of the mapping element is:

```
<mapping-module type="..."code="..." module="...">
  <module-option name="..." value="..."/>
</mapping-module>
```

The type attribute reflects the type of mapping of the provider and should be one of principal, credential, role or attribute. By default "role" is the type used if the attribute is not set.

The code attribute is used to specify the implementing class of the login module which can either be the full class name or one of the abbreviated names from the following list:

Code	Classname
PropertiesRoles	`org.jboss.security.mapping.providers.role.PropertiesRolesMappingP`
SimpleRoles	`org.jboss.security.mapping.providers.role.SimpleRolesMappingProvi`
DeploymentRoles	`org.jboss.security.mapping.providers.DeploymentRolesMappingProvid`
DatabaseRoles	`org.jboss.security.mapping.providers.role.DatabaseRolesMappingPro`
LdapRoles	`org.jboss.security.mapping.providers.role.LdapRolesMappingProvide`

The module attribute specifies the name of the JBoss Modules module from which the class specified by the code attribute should be loaded. Specifying it is not necessary if one of the abbreviated names in the above list is used.

The module-option element can be repeated zero or more times to specify the module options as required for the login module being configured. It requires the name and value attributes.

audit

The audit element can be used to define a custom audit provider. The default implementation used is `org.jboss.security.audit.providers.LogAuditProvider`. The structure of the audit element is:

```
<provider-module code="..." module="...">
  <module-option name="..." value="..."/>
</provider-module>
```

The code attribute is used to specify the implementing class of the provider module.

The module attribute specifies the name of the JBoss Modules module from which the class specified by the code attribute should be loaded.

The module-option element can be repeated zero or more times to specify the module options as required for the login module being configured. It requires the name and value attributes.

jsse

The jsse element defines configuration for keystores and truststores that can be used for SSL context configuration or for certificate storing/retrieving.

The set of attributes (all of them optional) of this element are:

keystore-password	Password of the keystore
keystore-type	Type of the keystore. By default it's "JKS"
keystore-url	`URL` where the keystore file can be found
keystore-provider	`Provider` of the keystore. The default JDK provider for the keystore type is used if this attribute is null
keystore-provider-argument	String that can be passed as the argument of the keystore `Provider` constructor
key-manager-factory-algorithm	Algorithm of the `KeyManagerFactory`. The default JDK algorithm of the key manager factory is used if this attribute is null
key-manager-factory-provider	`Provider` of the `KeyManagerFactory`. The default JDK provider for the key manager factory algorithm is used if this attribute is null
truststore-password	Password of the truststore
truststore-type	Type of the truststore. By deafult it's "JKS"
truststore-url	`URL` where the truststore file can be found
truststore-provider	`Provider` of the truststore. The default JDK provider for the truststore type is used if this attribute is null
truststore-provider-argument	String that can be passed as the argument of the truststore `Provider` constructor
trust-manager-factory-algorithm	Algorithm of the `TrustManagerFactory`. The default JDK algorithm of the trust manager factory is used if this attribute is null
trust-manager-factory-provider	`Provider` of the `TrustManagerFactory`. The default JDK provider for the trust manager factory algorithm is used if this attribute is null
client-alias	Alias of the keystore to be used when creating client side SSL sockets
server-alias	Alias of the keystore to be used when creating server side SSL sockets
service-auth-token	Validation token to enable third party services to retrieve a keystore `Key`. This is typically used to retrieve a private key for signing purposes
client-auth	Flag to indicate if the server side SSL socket should require client authentication. Default is "false"
cipher-suites	Comma separated list of cipher suites to be used by a `SSLContext`
protocols	Comma separated list of SSL protocols to be used by a `SSLContext`

The optional additional-properties element can be used to include other options. The structure of the jsse element is:

```
<jsse keystore-url="..." keystore-password="..." keystore-type="..." keystore-provider="..."
keystore-provider-argument="..." key-manager-factory-algorithm="..."
key-manager-factory-provider="..." truststore-url="..." truststore-password="..."
truststore-type="..." truststore-provider="..." truststore-provider-argument="..."
trust-manager-factory-algorithm="..." trust-manager-factory-provider="..." client-alias="..."
server-alias="..." service-auth-token="..." client-auth="..." cipher-suites="..."
protocols="...">
  <additional-properties>x=y
    a=b
  </additional-properties>
</jsse>
```

security-properties

This element is used to specify additional properties as required by the security subsystem, properties are specified in the following format:

```
<security-properties>
  <property name="..." value="..."/>
</security-properties>
```

The property element can be repeated as required for as many properties need to be defined.

Each property specified is set on the `java.security.Security` class.

5.22.8 Web services

JBossWS components are provided to the application server through the webservices subsystem. JBossWS components handle the processing of WS endpoints. The subsystem supports the configuration of published endpoint addresses, and endpoint handler chains. A default webservice subsystem is provided in the server's domain and standalone configuration files.

Structure of the webservices subsystem

Published endpoint address

JBossWS supports the rewriting of the `<soap:address>` element of endpoints published in WSDL contracts. This feature is useful for controlling the server address that is advertised to clients for each endpoint.

The following elements are available and can be modified (all are optional):

Name	Type	Description

modify-wsdl-address	boolean	This boolean enables and disables the address rewrite functionality. When modify-wsdl-address is set to true and the content of <soap:address> is a valid URL, JBossWS will rewrite the URL using the values of wsdl-host and wsdl-port or wsdl-secure-port. When modify-wsdl-address is set to false and the content of <soap:address> is a valid URL, JBossWS will not rewrite the URL. The <soap:address> URL will be used. When the content of <soap:address> is not a valid URL, JBossWS will rewrite it no matter what the setting of modify-wsdl-address. If modify-wsdl-address is set to true and wsdl-host is not defined or explicitly set to '`jbossws.undefined.host`' the content of <soap:address> URL is use. JBossWS uses the requester's host when rewriting the <soap:address> When modify-wsdl-address is not defined JBossWS uses a default value of true.
wsdl-host	string	The hostname / IP address to be used for rewriting `<soap:address>`. If `wsdl-host` is set to `jbossws.undefined.host`, JBossWS uses the requester's host when rewriting the `<soap:address>` When wsdl-host is not defined JBossWS uses a default value of '`jbossws.undefined.host`'.
wsdl-port	int	Set this property to explicitly define the HTTP port that will be used for rewriting the SOAP address. Otherwise the HTTP port will be identified by querying the list of installed HTTP connectors.
wsdl-secure-port	int	Set this property to explicitly define the HTTPS port that will be used for rewriting the SOAP address. Otherwise the HTTPS port will be identified by querying the list of installed HTTPS connectors.
wsdl-uri-scheme	string	This property explicitly sets the URI scheme to use for rewriting `<soap:address>` . Valid values are `http` and `https`. This configuration overrides scheme computed by processing the endpoint (even if a transport guarantee is specified). The provided values for `wsdl-port` and `wsdl-secure-port` (or their default values) are used depending on specified scheme.

wsdl-path-rewrite-rule	string	This string defines a SED substitution command (e.g., 's/regexp/replacement/g') that JBossWS executes against the path component of each <soap:address> URL published from the server. When wsdl-path-rewrite-rule is not defined, JBossWS retains the original path component of each <soap:address> URL. When 'modify-wsdl-address' is set to "false" this element is ignored.

Predefined endpoint configurations

JBossWS enables extra setup configuration data to be predefined and associated with an endpoint implementation. Predefined endpoint configurations can be used for JAX-WS client and JAX-WS endpoint setup. Endpoint configurations can include JAX-WS handlers and key/value properties declarations. This feature provides a convenient way to add handlers to WS endpoints and to set key/value properties that control JBossWS and Apache CXF internals (see Apache CXF configuration).

The webservices subsystem provides schema to support the definition of named sets of endpoint configuration data. Annotation, *org.jboss.ws.api.annotation.EndpointConfig* is provided to map the named configuration to the endpoint implementation.

There is no limit to the number of endpoint configurations that can be defined within the webservices subsystem. Each endpoint configuration must have a name that is unique within the webservices subsystem. Endpoint configurations defined in the webservices subsystem are available for reference by name through the annotation to any endpoint in a deployed application.

WildFly ships with two predefined endpoint configurations. Standard-Endpoint-Config is the default configuration. Recording-Endpoint-Config is an example of custom endpoint configuration and includes a recording handler.

```
[standalone@localhost:9999 /] /subsystem=webservices:read-resource
{
    "outcome" => "success",
    "result" => {
        "endpoint" => {},
        "modify-wsdl-address" => true,
        "wsdl-host" => expression "${jboss.bind.address:127.0.0.1}",
        "endpoint-config" => {
            "Standard-Endpoint-Config" => undefined,
            "Recording-Endpoint-Config" => undefined
        }
    }
}
```

The `Standard-Endpoint-Config` is a special endpoint configuration. It is used for any endpoint that does not have an explicitly assigned endpoint configuration.

Endpoint configs

Endpoint configs are defined using `the endpoint-config` element. Each endpoint configuration may include properties and handlers set to the endpoints associated to the configuration.

```
[standalone@localhost:9999 /]
/subsystem=webservices/endpoint-config=Recording-Endpoint-Config:read-resource
{
    "outcome" => "success",
    "result" => {
        "post-handler-chain" => undefined,
        "property" => undefined,
        "pre-handler-chain" => {"recording-handlers" => undefined}
    }
}
```

A new endpoint configuration can be added as follows:

```
[standalone@localhost:9999 /] /subsystem=webservices/endpoint-config=My-Endpoint-Config:add
{
    "outcome" => "success",
    "response-headers" => {
        "operation-requires-restart" => true,
        "process-state" => "restart-required"
    }
}
```

Handler chains

Each endpoint configuration may be associated with zero or more PRE and POST handler chains. Each handler chain may include JAXWS handlers. For outbound messages the PRE handler chains are executed before any handler that is attached to the endpoint using the standard means, such as with annotation @HandlerChain, and POST handler chains are executed after those objects have executed. For inbound messages the POST handler chains are executed before any handler that is attached to the endpoint using the standard means and the PRE handler chains are executed after those objects have executed.

```
* Server inbound messages
Client --> ... --> POST HANDLER --> ENDPOINT HANDLERS --> PRE HANDLERS --> Endpoint

* Server outbound messages
Endpoint --> PRE HANDLER --> ENDPOINT HANDLERS --> POST HANDLERS --> ... --> Client
```

The protocol-binding attribute must be used to set the protocols for which the chain will be triggered.

```
[standalone@localhost:9999 /]
/subsystem=webservices/endpoint-config=Recording-Endpoint-Config/pre-handler-chain=recording-handl
"outcome" => "success",
    "result" => {
        "protocol-bindings" => "##SOAP11_HTTP ##SOAP11_HTTP_MTOM ##SOAP12_HTTP
##SOAP12_HTTP_MTOM",
        "handler" => {"RecordingHandler" => undefined}
    },
    "response-headers" => {"process-state" => "restart-required"}
}
```

A new handler chain can be added as follows:

```
[standalone@localhost:9999 /]
/subsystem=webservices/endpoint-config=My-Endpoint-Config/post-handler-chain=my-handlers:add(proto
"outcome" => "success",
    "response-headers" => {
        "operation-requires-restart" => true,
        "process-state" => "restart-required"
    }
}
[standalone@localhost:9999 /]
/subsystem=webservices/endpoint-config=My-Endpoint-Config/post-handler-chain=my-handlers:read-reso
"outcome" => "success",
    "result" => {
        "handler" => undefined,
        "protocol-bindings" => "##SOAP11_HTTP"
    },
    "response-headers" => {"process-state" => "restart-required"}
}
```

Handlers

JAXWS handler can be added in handler chains:

```
[standalone@localhost:9999 /]
/subsystem=webservices/endpoint-config=Recording-Endpoint-Config/pre-handler-chain=recording-handl
"outcome" => "success",
    "result" => {"class" => "org.jboss.ws.common.invocation.RecordingServerHandler"},
    "response-headers" => {"process-state" => "restart-required"}
}
[standalone@localhost:9999 /]
/subsystem=webservices/endpoint-config=My-Endpoint-Config/post-handler-chain=my-handlers/handler=f
"outcome" => "success",
    "response-headers" => {
        "operation-requires-restart" => true,
        "process-state" => "restart-required"
    }
}
```

Endpoint-config handler classloading

The `class` attribute is used to provide the fully qualified class name of the handler. At deploy time, an instance of the class is created for each referencing deployment. For class creation to succeed, the deployment classloader must to be able to load the handler class.

Runtime information

Each web service endpoint is exposed through the deployment that provides the endpoint implementation. Each endpoint can be queried as a deployment resource. For further information please consult the chapter "Application Deployment". Each web service endpoint specifies a web context and a WSDL Url:

```
[standalone@localhost:9999 /] /deployment="*"/subsystem=webservices/endpoint="*":read-resource
{
   "outcome" => "success",
   "result" => [{
       "address" => [
           ("deployment" => "jaxws-samples-handlerchain.war"),
           ("subsystem" => "webservices"),
           ("endpoint" => "jaxws-samples-handlerchain:TestService")
       ],
       "outcome" => "success",
       "result" => {
           "class" => "org.jboss.test.ws.jaxws.samples.handlerchain.EndpointImpl",
           "context" => "jaxws-samples-handlerchain",
           "name" => "TestService",
           "type" => "JAXWS_JSE",
           "wsdl-url" => "http://localhost:8080/jaxws-samples-handlerchain?wsdl"
       }
   }]
}
```

Component Reference

The web service subsystem is provided by the JBossWS project. For a detailed description of the available configuration properties, please consult the project documentation.

- JBossWS homepage: http://www.jboss.org/jbossws
- Project Documentation: https://docs.jboss.org/author/display/JBWS

5.22.9 Resource adapters

Resource adapters are configured through the *resource-adapters* subsystem. Declaring a new resource adapter consists of two separate steps: You would need to deploy the .rar archive and define a resource adapter entry in the subsystem.

Resource Adapter Definitions

The resource adapter itself is defined within the subsystem *resource-adapters*:

```
<subsystem xmlns="urn:jboss:domain:resource-adapters:1.0">
    <resource-adapters>
       <resource-adapter>
          <archive>eis.rar</archive>
          <!-- Resource adapter level config-property -->
          <config-property name="Server">localhost</config-property>
          <config-property name="Port">19000</config-property>
          <transaction-support>XATransaction</transaction-support>
          <connection-definitions>
             <connection-definition class-name="com.acme.eis.ra.EISManagedConnectionFactory"
                                    jndi-name="java:/eis/AcmeConnectionFactory"
                                    pool-name="AcmeConnectionFactory">
                <!-- Managed connection factory level config-property -->
                <config-property name="Name">Acme Inc</config-property>
                <pool>
                   <min-pool-size>10</min-pool-size>
                   <max-pool-size>100</max-pool-size>
                </pool>
                <security>
                   <application/>
                </security>
             </connection-definition>
          </connection-definitions>
          <admin-objects>
              <admin-object class-name="com.acme.eis.ra.EISAdminObjectImpl"
                            jndi-name="java:/eis/AcmeAdminObject">
                 <config-property name="Threshold">10</config-property>
              </admin-object>
          </admin-objects>
       </resource-adapter>
    </resource-adapters>
</subsystem>
```

Note, that only JNDI bindings under java:/ or java:jboss/ are supported.

(See `standalone/configuration/standalone.xml`)

Using security domains

Information about using security domains can be found at
https://community.jboss.org/wiki/JBossAS7SecurityDomainModel

Automatic activation of resource adapter archives

A resource adapter archive can be automatically activated with a configuration by including an META-INF/ironjacamar.xml in the archive.

The schema can be found at http://docs.jboss.org/ironjacamar/schema/ironjacamar_1_0.xsd

Component Reference

The resource adapter subsystem is provided by the IronJacamar project. For a detailed description of the available configuration properties, please consult the project documentation.

- IronJacamar homepage: http://www.jboss.org/ironjacamar
- Project Documentation: http://www.jboss.org/ironjacamar/docs
- Schema description: http://docs.jboss.org/ironjacamar/userguide/1.0/en-US/html/deployment.html#deployingra_descriptor

5.22.10 Batch

Overview

The batch subsystem is used to configure an environment for running batch applications. WildFly uses JBeret for it's batch implementation. Specific information about JBeret can be found in the user guide. The resource path, in CLI notation, for the subsystem is `subsystem=batch-jberet`.

Default Subsystem Configuration

For up to date information about subsystem configuration options see http://wildscribe.github.io/.

Security

A new `security-domain` attribute was added to the `batch-jberet` subsystem to allow batch jobs to be executed under that security domain. Jobs that are stopped as part of a `suspend` operation will be restarted on execution of a `resume` with the original user that started job.

There was a `org.wildfly.extension.batch.jberet.deployment.BatchPermission` added to allow a security restraint to various batch functions. The following functions can be controlled with this permission.

- start
- stop
- restart
- abandon
- read

The read function allows users to use the getter methods from the `javax.batch.operations.JobOperator` or read the `batch-jberet` deployment resource, for example `/deployment=my.war/subsystem=batch-jberet:read-resource`.

Deployment Descriptors

There are no deployment descriptors for configuring a batch environment defined by the JSR-352 specification. In WildFly you can use a `jboss-all.xml` deployment descriptor to define aspects of the batch environment for your deployment.

In the `jboss-all.xml` deployment descriptor you can define a named job repository, a new job repository and/or a named thread pool. A named job repository and named thread pool are resources defined on the batch subsystem. Only a named thread pool is allowed to be defined in the deployment descriptor.

Example Named Job Repository and Thread Pool

```
<jboss umlns="urn:jboss:1.0">
    <batch xmlns="urn:jboss:batch-jberet:1.0">
      <job-repository>
        <named name="batch-ds"/>
      </job-repository>
      <thread-pool name="deployment-thread-pool"/>
    </batch>
</jboss>
```

Example new Job Repository

```
<jboss umlns="urn:jboss:1.0">
    <batch xmlns="urn:jboss:batch-jberet:1.0">
        <job-repository>
            <jdbc jndi-name="java:jboss/datasources/ExampleDS"/>
        </job-repository>
    </batch>
</jboss>
```

Deployment Resources

Some subsystems in WildFly register runtime resources for deployments. The batch subsystem registers jobs and executions. The jobs are registered using the job name, this is *not* the job XML name. Executions are registered using the execution id.

Batch application in a standalone server

```
[standalone@localhost:9990 /]
/deployment=batch-jdbc-chunk.war/subsystem=batch-jberet:read-resource(recursive=true,include-runti
"outcome" => "success",
    "result" => {"job" => {
        "reader-3" => {
            "instance-count" => 1,
            "running-executions" => 0,
            "execution" => {"1" => {
                "batch-status" => "COMPLETED",
                "create-time" => "2015-08-07T15:37:06.416-0700",
                "end-time" => "2015-08-07T15:37:06.519-0700",
                "exit-status" => "COMPLETED",
                "instance-id" => 1L,
                "last-updated-time" => "2015-08-07T15:37:06.519-0700",
                "start-time" => "2015-08-07T15:37:06.425-0700"
            }}
        },
        "reader-5" => {
            "instance-count" => 0,
            "running-executions" => 0,
            "execution" => undefined
        }
    }}
}
```

The batch subsystem resource on a deployment also has 3 operations to interact with batch jobs on the selected deployment. There is a `start-job`, `stop-job` and `restart-job` operation. The `execution` resource also has a `stop-job` and `restart-job` operation.

Example start-job

```
[standalone@localhost:9990 /]
/deployment=batch-chunk.war/subsystem=batch-jberet:start-job(job-xml-name=simple,
properties={writer.sleep=5000})
{
    "outcome" => "success",
    "result" => 1L
}
```

Example stop-job

```
[standalone@localhost:9990 /]
/deployment=batch-chunk.war/subsystem=batch-jberet:stop-job(execution-id=2)
```

Example restart-job

```
[standalone@localhost:9990 /]
/deployment=batch-chunk.war/subsystem=batch-jberet:restart-job(execution-id=2)
{
    "outcome" => "success",
    "result" => 3L
}
```

Result of resource after the 3 executions

```
[standalone@localhost:9990 /]
/deployment=batch-chunk.war/subsystem=batch-jberet:read-resource(recursive=true,
include-runtime=true)
{
    "outcome" => "success",
    "result" => {"job" => {"chunkPartition" => {
        "instance-count" => 2,
        "running-executions" => 0,
        "execution" => {
            "1" => {
                "batch-status" => "COMPLETED",
                "create-time" => "2015-08-07T15:41:55.504-0700",
                "end-time" => "2015-08-07T15:42:15.513-0700",
                "exit-status" => "COMPLETED",
                "instance-id" => 1L,
                "last-updated-time" => "2015-08-07T15:42:15.513-0700",
                "start-time" => "2015-08-07T15:41:55.504-0700"
            },
            "2" => {
                "batch-status" => "STOPPED",
                "create-time" => "2015-08-07T15:44:39.879-0700",
                "end-time" => "2015-08-07T15:44:54.882-0700",
                "exit-status" => "STOPPED",
                "instance-id" => 2L,
                "last-updated-time" => "2015-08-07T15:44:54.882-0700",
                "start-time" => "2015-08-07T15:44:39.879-0700"
            },
            "3" => {
                "batch-status" => "COMPLETED",
                "create-time" => "2015-08-07T15:45:48.162-0700",
                "end-time" => "2015-08-07T15:45:53.165-0700",
                "exit-status" => "COMPLETED",
                "instance-id" => 2L,
                "last-updated-time" => "2015-08-07T15:45:53.165-0700",
                "start-time" => "2015-08-07T15:45:48.163-0700"
            }
        }
    }}}
}
```

Pro Tip

You can filter jobs by an attribute on the execution resource with the `query` operation.

View all stopped jobs

```
/deployment=batch-chunk.war/subsystem=batch-jberet/job=*/execution=*:query(where=["batch-sta
"STOPPED"])
```

As with all operations you can see details about the operation using the `:read-operation-description` operation.

Tab completion

Don't forget that CLI has tab completion which will complete operations and attributes (arguments) on operations.

Example start-job operation description

```
[standalone@localhost:9990 /]
/deployment=batch-chunk.war/subsystem=batch-jberet:read-operation-description(name=start-job)
{
    "outcome" => "success",
    "result" => {
        "operation-name" => "start-job",
        "description" => "Starts a batch job.",
        "request-properties" => {
            "job-xml-name" => {
                "type" => STRING,
                "description" => "The name of the job XML file to use when starting the job.",
                "expressions-allowed" => false,
                "required" => true,
                "nillable" => false,
                "min-length" => 1L,
                "max-length" => 2147483647L
            },
            "properties" => {
                "type" => OBJECT,
                "description" => "Optional properties to use when starting the batch job.",
                "expressions-allowed" => false,
                "required" => false,
                "nillable" => true,
                "value-type" => STRING
            }
        },
        "reply-properties" => {"type" => LONG},
        "read-only" => false,
        "runtime-only" => true
    }
}
```

5.22.11 JSF

Overview

JSF configuration is handled by the JSF subsystem. The JSF subsystem allows multiple JSF implementations to be installed on the same WildFly server. In particular, any version of Mojarra or MyFaces that implements spec level 2.1 or higher can be installed. For each JSF implementation, a new slot needs to be created under `com.sun.jsf-impl`, `javax.faces.api`, and `org.jboss.as.jsf-injection`. When the JSF subsystem starts up, it scans the module path to find all of the JSF implementations that have been installed. The default JSF implementation that WildFly should use is defined by the `default-jsf-impl-slot` attribute.

Installing a new JSF implementation manually

A new JSF implementation can be manually installed as follows:

Add a module slot for the new JSF implementation JAR

- Create the following directory structure under the WILDFLY_HOME/modules directory:
  WILDFLY_HOME/modules/com/sun/jsf-impl/<JSF_IMPL_NAME>-<JSF_VERSION>

  For example, for Mojarra 2.2.11, the above path would resolve to:
  WILDFLY_HOME/modules/com/sun/jsf-impl/mojarra-2.2.11

- Place the JSF implementation JAR in the <JSF_IMPL_NAME>-<JSF_VERSION> subdirectory. In the same subdirectory, add a `module.xml` file similar to the Mojarra or MyFaces template examples. Change the `resource-root-path` to the name of your JSF implementation JAR and fill in appropriate values for ${`jsf-impl-name`} and ${`jsf-version`}.

Add a module slot for the new JSF API JAR

- Create the following directory structure under the WILDFLY_HOME/modules directory:
  WILDFLY_HOME/modules/javax/faces/api/<JSF_IMPL_NAME>-<JSF_VERSION>

- Place the JSF API JAR in the <JSF_IMPL_NAME>-<JSF_VERSION> subdirectory. In the same subdirectory, add a `module.xml` file similar to the Mojarra or MyFaces template examples. Change the `resource-root-path` to the name of your JSF API JAR and fill in appropriate values for ${`jsf-impl-name`} and ${`jsf-version`}.

Add a module slot for the JSF injection JAR

- Create the following directory structure under the WILDFLY_HOME/modules directory: WILDFLY_HOME/modules/org/jboss/as/jsf-injection/<JSF_IMPL_NAME>-<JSF_VERSION>
- Copy the wildfly-jsf-injection JAR and the weld-core-jsf JAR from WILDFLY_HOME/modules/system/layers/base/org/jboss/as/jsf-injection/main to the <JSF_IMPL_NAME>-<JSF_VERSION> subdirectory.
- In the <JSF_IMPL_NAME>-<JSF_VERSION> subdirectory, add a `module.xml` file similar to the Mojarra or MyFaces template examples and fill in appropriate values for ${`jsf-impl-name`}, ${`jsf-version`}, ${`version.jboss.as`}, and ${`version.weld.core`}. (These last two placeholders depend on the versions of the wildfly-jsf-injection and weld-core-jsf JARs that were copied over in the previous step.)

For MyFaces only - add a module for the commons-digester JAR

- Create the following directory structure under the WILDFLY_HOME/modules directory: WILDFLY_HOME/modules/org/apache/commons/digester/main
- Place the commons-digester JAR in WILDFLY_HOME/modules/org/apache/commons/digester/main. In the `main` subdirectory, add a `module.xml` file similar to this template. Fill in the appropriate value for ${`version.commons-digester`}.

Start the server

After starting the server, the following CLI command can be used to verify that your new JSF implementation has been installed successfully. The new JSF implementation should appear in the output of this command.

```
[standalone@localhost:9990 /] /subsystem=jsf:list-active-jsf-impls()
```

Changing the default JSF implementation

The following CLI command can be used to make a newly installed JSF implementation the default JSF implementation used by WildFly:

```
/subsystem=jsf:write-attribute(name=default-jsf-impl-slot,value=<JSF_IMPL_NAME>-<JSF_VERSION>)
```

A server restart will be required for this change to take effect.

Configuring a JSF app to use a non-default JSF implementation

A JSF app can be configured to use an installed JSF implementation that's not the default implementation by adding a `org.jboss.jbossfaces.JSF_CONFIG_NAME` context parameter to its `web.xml` file. For example, to indicate that a JSF app should use MyFaces 2.2.12 (assuming MyFaces 2.2.12 has been installed on the server), the following context parameter would need to be added:

```
<context-param>
  <param-name>org.jboss.jbossfaces.JSF_CONFIG_NAME</param-name>
  <param-value>myfaces-2.2.12</param-value>
</context-param>
```

If a JSF app does not specify this context parameter, the default JSF implementation will be used for that app.

5.22.12 JMX

The JMX subsystem registers a service with the Remoting endpoint so that remote access to JMX can be obtained over the exposed Remoting connector.

This is switched on by default in standalone mode and accessible over port 9990 but in domain mode is switched off so needs to be enabled - in domain mode the port will be the port of the Remoting connector for the WildFly instance to be monitored.

To use the connector you can access it in the standard way using a `service:jmx` URL:

```
import javax.management.MBeanServerConnection;
import javax.management.remote.JMXConnector;
import javax.management.remote.JMXConnectorFactory;
import javax.management.remote.JMXServiceURL;

public class JMXExample {

    public static void main(String[] args) throws Exception {
        //Get a connection to the WildFly MBean server on localhost
        String host = "localhost";
        int port = 9990;  // management-web port
        String urlString =
            System.getProperty("jmx.service.url","service:jmx:remote+http://" + host + ":" +
port);
        JMXServiceURL serviceURL = new JMXServiceURL(urlString);
        JMXConnector jmxConnector = JMXConnectorFactory.connect(serviceURL, null);
        MBeanServerConnection connection = jmxConnector.getMBeanServerConnection();

        //Invoke on the WildFly MBean server
        int count = connection.getMBeanCount();
        System.out.println(count);
        jmxConnector.close();
    }
}
```

You also need to set your classpath when running the above example. The following script covers Linux. If your environment is much different, paste your script when you have it working.

```
!/bin/bash

# specify your WildFly folder
export YOUR_JBOSS_HOME=~/WildFly

java -classpath $YOUR_JBOSS_HOME/bin/client/jboss-client.jar:./ JMXExample
```

You can also connect using jconsole.

If using jconsole use the `jconsole.sh` and `jconsole.bat` scripts included in the /bin directory of the WildFly distribution as these set the classpath as required to connect over Remoting.

In addition to the standard JVM MBeans, the WildFly MBean server contains the following MBeans:

JMX ObjectName	Description
`jboss.msc:type=container,name=jboss-as`	Exposes management operations on the JBoss Modular Service Container, which is the dependency injection framework at the heart of WildFly. It is useful for debugging dependency problems, for example if you are integrating your own subsystems, as it exposes operations to dump all services and their current states
`jboss.naming:type=JNDIView`	Shows what is bound in JNDI
`jboss.modules:type=ModuleLoader,name=*`	This collection of MBeans exposes management operations on JBoss Modules classloading layer. It is useful for debugging dependency problems arising from missing module dependencies

Audit logging

Audit logging for the JMX MBean server managed by the JMX subsystem. The resource is at `/subsystem=jmx/configuration=audit-log` and its attributes are similar to the ones mentioned for `/core-service=management/access=audit/logger=audit-log` in Audit logging.

Attribute	Description
`enabled`	`true` to enable logging of the JMX operations
`log-boot`	`true` to log the JMX operations when booting the server, `false` otherwise
`log-read-only`	If `true` all operations will be audit logged, if `false` only operations that change the model will be logged

Then which handlers are used to log the management operations are configured as `handler=*` children of the logger. These handlers and their formatters are defined in the global `/core-service=management/access=audit` section mentioned in Audit logging.

JSON Formatter

The same JSON Formatter is used as described in Audit logging. However the records for MBean Server invocations have slightly different fields from those logged for the core management layer.

```
2013-08-29 18:26:29 - {
    "type" : "jmx",
    "r/o" : false,
    "booting" : false,
    "version" : "10.0.0.Final",
    "user" : "$local",
    "domainUUID" : null,
    "access" : "JMX",
    "remote-address" : "127.0.0.1/127.0.0.1",
    "method" : "invoke",
    "sig" : [
        "javax.management.ObjectName",
        "java.lang.String",
        "[Ljava.lang.Object;",
        "[Ljava.lang.String;"
    ],
    "params" : [
        "java.lang:type=Threading",
        "getThreadInfo",
        "[Ljava.lang.Object;@5e6c33c",
        "[Ljava.lang.String;@4b681c69"
    ]
}
```

It includes an optional timestamp and then the following information in the json record

Field name	Description
`type`	This will have the value `jmx` meaning it comes from the jmx subsystem
`r/o`	`true` if the operation has read only impact on the MBean(s)
`booting`	`true` if the operation was executed during the bootup process, `false` if it was executed once the server is up and running
`version`	The version number of the WildFly instance
`user`	The username of the authenticated user.
`domainUUID`	This is not currently populated for JMX operations
`access`	This can have one of the following values: *`NATIVE` - The operation came in through the native management interface, for example the CLI *`HTTP` - The operation came in through the domain HTTP interface, for example the admin console *`JMX` - The operation came in through the JMX subsystem. See JMX for how to configure audit logging for JMX.
`remote-address`	The address of the client executing this operation
`method`	The name of the called MBeanServer method
`sig`	The signature of the called called MBeanServer method
`params`	The actual parameters passed in to the MBeanServer method, a simple `Object.toString()` is called on each parameter.
`error`	If calling the MBeanServer method resulted in an error, this field will be populated with `Throwable.getMessage()`

5.22.13 Deployment Scanner

The deployment scanner is only used in standalone mode. Its job is to monitor a directory for new files and to deploy those files. It can be found in `standalone.xml`:

```
<subsystem xmlns="urn:jboss:domain:deployment-scanner:2.0">
   <deployment-scanner scan-interval="5000"
      relative-to="jboss.server.base.dir" path="deployments" />
</subsystem>
```

You can define more `deployment-scanner` entries to scan for deployments from more locations. The configuration showed will scan the `JBOSS_HOME/standalone/deployments` directory every five seconds. The runtime model is shown below, and uses default values for attributes not specified in the xml:

```
[standalone@localhost:9999 /] /subsystem=deployment-scanner:read-resource(recursive=true)
{
    "outcome" => "success",
    "result" => {"scanner" => {"default" => {
        "auto-deploy-exploded" => false,
        "auto-deploy-zipped" => true,
        "deployment-timeout" => 60L,
        "name" => "default",
        "path" => "deployments",
        "relative-to" => "jboss.server.base.dir",
        "scan-enabled" => true,
        "scan-interval" => 5000
    }}}
}
```

The attributes are

Name	Type	Description
`name`	STRING	The name of the scanner. `default` is used if not specified
`path`	STRING	The actual filesystem path to be scanned. Treated as an absolute path, unless the 'relative-to' attribute is specified, in which case the value is treated as relative to that path.
`relative-to`	STRING	Reference to a filesystem path defined in the "paths" section of the server configuration, or one of the system properties specified on startup. In the example above `jboss.server.base.dir` resolves to `JBOSS_HOME/standalone`
`scan-enabled`	BOOLEAN	If true scanning is enabled
`scan-interval`	INT	Periodic interval, in milliseconds, at which the repository should be scanned for changes. A value of less than 1 indicates the repository should only be scanned at initial startup.
`auto-deploy-zipped`	BOOLEAN	Controls whether zipped deployment content should be automatically deployed by the scanner without requiring the user to add a .dodeploy marker file.
`auto-deploy-exploded`	BOOLEAN	Controls whether exploded deployment content should be automatically deployed by the scanner without requiring the user to add a .dodeploy marker file. Setting this to 'true' is not recommended for anything but basic development scenarios, as there is no way to ensure that deployment will not occur in the middle of changes to the content.
`auto-deploy-xml`	BOOLEAN	Controls whether XML content should be automatically deployed by the scanner without requiring a .dodeploy marker file.
`deployment-timeout`	LONG	Timeout, in seconds, a deployment is allows to execute before being canceled. The default is 60 seconds.

Deployment scanners can be added by modifying `standalone.xml` before starting up the server or they can be added and removed at runtime using the CLI

```
[standalone@localhost:9990 /]
/subsystem=deployment-scanner/scanner=new:add(scan-interval=10000,relative-to="jboss.server.base.d
=> "success"}
[standalone@localhost:9990 /] /subsystem=deployment-scanner/scanner=new:remove
{"outcome" => "success"}
```

You can also change the attributes at runtime, so for example to turn off scanning you can do

```
[standalone@localhost:9990 /]
/subsystem=deployment-scanner/scanner=default:write-attribute(name="scan-enabled",value=false)
{"outcome" => "success"}
[standalone@localhost:9990 /] /subsystem=deployment-scanner:read-resource(recursive=true)
{
    "outcome" => "success",
    "result" => {"scanner" => {"default" => {
        "auto-deploy-exploded" => false,
        "auto-deploy-zipped" => true,
        "deployment-timeout" => 60L,
        "name" => "default",
        "path" => "deployments",
        "relative-to" => "jboss.server.base.dir",
        "scan-enabled" => false,
        "scan-interval" => 5000
    }}}
}
```

5.22.14 Core Management

Overview

The core management subsystem is composed services used to manage the server or monitor its status. The core management subsystem configuration may be used to:

- register a listener for a server lifecycle events.
- list the last configuration changes on a server.

Lifecycle listener

You can create an implementation of *org.wildfly.extension.core.management.client.ProcessStateListener* which will be notified on running and runtime configuration state changes thus enabling the developer to react to those changes.

In order to use this feature you need to create your own module then configure and deploy it using the core management subsystem.

For example let's create a simple listener :

```
public class SimpleListener implements ProcessStateListener {

    private File file;
    private FileWriter fileWriter;
    private ProcessStateListenerInitParameters parameters;

    @Override
    public void init(ProcessStateListenerInitParameters parameters) {
        this.parameters = parameters;
        this.file = new File(parameters.getInitProperties().get("file"));
        try {
            fileWriter = new FileWriter(file, true);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void cleanup() {
        try {
            fileWriter.close();
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            fileWriter = null;
        }
    }

    @Override
    public void runtimeConfigurationStateChanged(RuntimeConfigurationStateChangeEvent evt) {
        try {
            fileWriter.write(String.format("%s %s %s %s\n", parameters.getProcessType(),
parameters.getRunningMode(), evt.getOldState(), evt.getNewState()));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void runningStateChanged(RunningStateChangeEvent evt) {
        try {
            fileWriter.write(String.format("%s %s %s %s\n", parameters.getProcessType(),
parameters.getRunningMode(), evt.getOldState(), evt.getNewState()));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

To compile it you need to depend on the *org.wildfly.core:wildfly-core-management-client* maven module.
Now let's add the module to the wildfly modules :

```
module add --name=org.simple.lifecycle.events.listener
--dependencies=org.wildfly.extension.core-management-client
--resources=/home/ehsavoie/dev/demo/simple-listener/target/simple-process-state-listener.jar
```

Now we can register or listener :

```
/subsystem=core-management/process-state-listener=simple-listener:add(class=org.simple.lifecycle.e
module=org.simple.lifecycle.events.listener, properties={file=/home/wildfly/tmp/events.txt})
```

Configuration changes

You can use the core management subsystem to enable and configure an **in-memory** history of the last configuration changes.
For example to track the last 5 configuration changes let's active this :

```
/subsystem=core-management/service=configuration-changes:add(max-history=5)
```

Now we can list the last configuration changes :

```
/subsystem=core-management/service=configuration-changes:list-changes()
{
    "outcome" => "success",
    "result" => [{
        "operation-date" => "2016-12-05T11:05:12.867Z",
        "access-mechanism" => "NATIVE",
        "remote-address" => "/127.0.0.1",
        "outcome" => "success",
        "operations" => [{
            "address" => [
                ("subsystem" => "core-management"),
                ("service" => "configuration-changes")
            ],
            "operation" => "add",
            "max-history" => 5,
            "operation-headers" => {
                "caller-type" => "user",
                "access-mechanism" => "NATIVE"
            }
        }]
    }]
}
```

5.22.15 Simple configuration subsystems

The following subsystems currently have no configuration beyond its root element in the configuration

```
<subsystem xmlns="urn:jboss:domain:jaxrs:1.0"/>
<subsystem xmlns="urn:jboss:domain:jdr:1.0"/>
<subsystem xmlns="urn:jboss:domain:pojo:1.0"/>
<subsystem xmlns="urn:jboss:domain:sar:1.0"/>
```

The presence of each of these turns on a piece of functionality:

Name	Description
`jaxrs`	Enables the deployment and functionality of JAX-RS applications
`jdr`	Enables the gathering of diagnostic data for use in remote analysis of error conditions. Although the data is in a simple format and could be useful to anyone, primarily useful for JBoss EAP subscribers who would provide the data to Red Hat when requesting support
`pojo`	Enables the deployment of applications containing JBoss Microcontainer services, as supported by previous versions of JBoss Application Server
`sar`	Enables the deployment of .SAR archives containing MBean services, as supported by previous versions of JBoss Application Server

5.22.16 Batch (JSR-352) Subsystem Configuration

Overview

The batch subsystem is used to configure an environment for running batch applications. WildFly uses JBeret for it's batch implementation. Specific information about JBeret can be found in the user guide. The resource path, in CLI notation, for the subsystem is `subsystem=batch-jberet`.

Default Subsystem Configuration

For up to date information about subsystem configuration options see http://wildscribe.github.io/.

Security

A new `security-domain` attribute was added to the `batch-jberet` subsystem to allow batch jobs to be executed under that security domain. Jobs that are stopped as part of a `suspend` operation will be restarted on execution of a `resume` with the original user that started job.

There was a `org.wildfly.extension.batch.jberet.deployment.BatchPermission` added to allow a security restraint to various batch functions. The following functions can be controlled with this permission.

- start
- stop
- restart
- abandon
- read

The read function allows users to use the getter methods from the `javax.batch.operations.JobOperator` or read the `batch-jberet` deployment resource, for example `/deployment=my.war/subsystem=batch-jberet:read-resource`.

Deployment Descriptors

There are no deployment descriptors for configuring a batch environment defined by the JSR-352 specification. In WildFly you can use a `jboss-all.xml` deployment descriptor to define aspects of the batch environment for your deployment.

In the `jboss-all.xml` deployment descriptor you can define a named job repository, a new job repository and/or a named thread pool. A named job repository and named thread pool are resources defined on the batch subsystem. Only a named thread pool is allowed to be defined in the deployment descriptor.

Example Named Job Repository and Thread Pool

```
<jboss umlns="urn:jboss:1.0">
    <batch xmlns="urn:jboss:batch-jberet:1.0">
      <job-repository>
        <named name="batch-ds"/>
      </job-repository>
      <thread-pool name="deployment-thread-pool"/>
    </batch>
</jboss>
```

Example new Job Repository

```
<jboss umlns="urn:jboss:1.0">
    <batch xmlns="urn:jboss:batch-jberet:1.0">
        <job-repository>
            <jdbc jndi-name="java:jboss/datasources/ExampleDS"/>
        </job-repository>
    </batch>
</jboss>
```

Deployment Resources

Some subsystems in WildFly register runtime resources for deployments. The batch subsystem registers jobs and executions. The jobs are registered using the job name, this is *not* the job XML name. Executions are registered using the execution id.

Batch application in a standalone server

```
[standalone@localhost:9990 /]
/deployment=batch-jdbc-chunk.war/subsystem=batch-jberet:read-resource(recursive=true,include-runti
"outcome" => "success",
    "result" => {"job" => {
        "reader-3" => {
            "instance-count" => 1,
            "running-executions" => 0,
            "execution" => {"1" => {
                "batch-status" => "COMPLETED",
                "create-time" => "2015-08-07T15:37:06.416-0700",
                "end-time" => "2015-08-07T15:37:06.519-0700",
                "exit-status" => "COMPLETED",
                "instance-id" => 1L,
                "last-updated-time" => "2015-08-07T15:37:06.519-0700",
                "start-time" => "2015-08-07T15:37:06.425-0700"
            }}
        },
        "reader-5" => {
            "instance-count" => 0,
            "running-executions" => 0,
            "execution" => undefined
        }
    }}
}
```

The batch subsystem resource on a deployment also has 3 operations to interact with batch jobs on the selected deployment. There is a `start-job`, `stop-job` and `restart-job` operation. The `execution` resource also has a `stop-job` and `restart-job` operation.

Example start-job

```
[standalone@localhost:9990 /]
/deployment=batch-chunk.war/subsystem=batch-jberet:start-job(job-xml-name=simple,
properties={writer.sleep=5000})
{
    "outcome" => "success",
    "result" => 1L
}
```

Example stop-job

```
[standalone@localhost:9990 /]
/deployment=batch-chunk.war/subsystem=batch-jberet:stop-job(execution-id=2)
```

Example restart-job

```
[standalone@localhost:9990 /]
/deployment=batch-chunk.war/subsystem=batch-jberet:restart-job(execution-id=2)
{
    "outcome" => "success",
    "result" => 3L
}
```

Result of resource after the 3 executions

```
[standalone@localhost:9990 /]
/deployment=batch-chunk.war/subsystem=batch-jberet:read-resource(recursive=true,
include-runtime=true)
{
    "outcome" => "success",
    "result" => {"job" => {"chunkPartition" => {
        "instance-count" => 2,
        "running-executions" => 0,
        "execution" => {
            "1" => {
                "batch-status" => "COMPLETED",
                "create-time" => "2015-08-07T15:41:55.504-0700",
                "end-time" => "2015-08-07T15:42:15.513-0700",
                "exit-status" => "COMPLETED",
                "instance-id" => 1L,
                "last-updated-time" => "2015-08-07T15:42:15.513-0700",
                "start-time" => "2015-08-07T15:41:55.504-0700"
            },
            "2" => {
                "batch-status" => "STOPPED",
                "create-time" => "2015-08-07T15:44:39.879-0700",
                "end-time" => "2015-08-07T15:44:54.882-0700",
                "exit-status" => "STOPPED",
                "instance-id" => 2L,
                "last-updated-time" => "2015-08-07T15:44:54.882-0700",
                "start-time" => "2015-08-07T15:44:39.879-0700"
            },
            "3" => {
                "batch-status" => "COMPLETED",
                "create-time" => "2015-08-07T15:45:48.162-0700",
                "end-time" => "2015-08-07T15:45:53.165-0700",
                "exit-status" => "COMPLETED",
                "instance-id" => 2L,
                "last-updated-time" => "2015-08-07T15:45:53.165-0700",
                "start-time" => "2015-08-07T15:45:48.163-0700"
            }
        }
    }}}
}
```

Pro Tip

You can filter jobs by an attribute on the execution resource with the `query` operation.

View all stopped jobs

```
/deployment=batch-chunk.war/subsystem=batch-jberet/job=*/execution=*:query(where=["batch-sta
"STOPPED"])
```

As with all operations you can see details about the operation using the `:read-operation-description` operation.

Tab completion

Don't forget that CLI has tab completion which will complete operations and attributes (arguments) on operations.

Example start-job operation description

```
[standalone@localhost:9990 /]
/deployment=batch-chunk.war/subsystem=batch-jberet:read-operation-description(name=start-job)
{
    "outcome" => "success",
    "result" => {
        "operation-name" => "start-job",
        "description" => "Starts a batch job.",
        "request-properties" => {
            "job-xml-name" => {
                "type" => STRING,
                "description" => "The name of the job XML file to use when starting the job.",
                "expressions-allowed" => false,
                "required" => true,
                "nillable" => false,
                "min-length" => 1L,
                "max-length" => 2147483647L
            },
            "properties" => {
                "type" => OBJECT,
                "description" => "Optional properties to use when starting the batch job.",
                "expressions-allowed" => false,
                "required" => false,
                "nillable" => true,
                "value-type" => STRING
            }
        },
        "reply-properties" => {"type" => LONG},
        "read-only" => false,
        "runtime-only" => true
    }
}
```

5.22.17 Core Management Subsystem Configuration

Overview

The core management subsystem is composed services used to manage the server or monitor its status. The core management subsystem configuration may be used to:

- register a listener for a server lifecycle events.
- list the last configuration changes on a server.

Lifecycle listener

You can create an implementation of *org.wildfly.extension.core.management.client.ProcessStateListener* which will be notified on running and runtime configuration state changes thus enabling the developer to react to those changes.

In order to use this feature you need to create your own module then configure and deploy it using the core management subsystem.

For example let's create a simple listener :

```
public class SimpleListener implements ProcessStateListener {

    private File file;
    private FileWriter fileWriter;
    private ProcessStateListenerInitParameters parameters;

    @Override
    public void init(ProcessStateListenerInitParameters parameters) {
        this.parameters = parameters;
        this.file = new File(parameters.getInitProperties().get("file"));
        try {
            fileWriter = new FileWriter(file, true);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void cleanup() {
        try {
            fileWriter.close();
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            fileWriter = null;
        }
    }

    @Override
    public void runtimeConfigurationStateChanged(RuntimeConfigurationStateChangeEvent evt) {
        try {
            fileWriter.write(String.format("%s %s %s %s\n", parameters.getProcessType(),
parameters.getRunningMode(), evt.getOldState(), evt.getNewState()));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void runningStateChanged(RunningStateChangeEvent evt) {
        try {
            fileWriter.write(String.format("%s %s %s %s\n", parameters.getProcessType(),
parameters.getRunningMode(), evt.getOldState(), evt.getNewState()));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

To compile it you need to depend on the *org.wildfly.core:wildfly-core-management-client* maven module.
Now let's add the module to the wildfly modules :

```
module add --name=org.simple.lifecycle.events.listener
--dependencies=org.wildfly.extension.core-management-client
--resources=/home/ehsavoie/dev/demo/simple-listener/target/simple-process-state-listener.jar
```

Now we can register or listener :

```
/subsystem=core-management/process-state-listener=simple-listener:add(class=org.simple.lifecycle.e
module=org.simple.lifecycle.events.listener, properties={file=/home/wildfly/tmp/events.txt})
```

Configuration changes

You can use the core management subsystem to enable and configure an **in-memory** history of the last configuration changes.
For example to track the last 5 configuration changes let's active this :

```
/subsystem=core-management/service=configuration-changes:add(max-history=5)
```

Now we can list the last configuration changes :

```
/subsystem=core-management/service=configuration-changes:list-changes()
{
    "outcome" => "success",
    "result" => [{
        "operation-date" => "2016-12-05T11:05:12.867Z",
        "access-mechanism" => "NATIVE",
        "remote-address" => "/127.0.0.1",
        "outcome" => "success",
        "operations" => [{
            "address" => [
                ("subsystem" => "core-management"),
                ("service" => "configuration-changes")
            ],
            "operation" => "add",
            "max-history" => 5,
            "operation-headers" => {
                "caller-type" => "user",
                "access-mechanism" => "NATIVE"
            }
        }]
    }]
}
```

5.22.18 DataSource configuration

Datasources are configured through the *datasource* subsystem. Declaring a new datasource consists of two separate steps: You would need to provide a JDBC driver and define a datasource that references the driver you installed.

JDBC Driver Installation

The recommended way to install a JDBC driver into WildFly 8 is to deploy it as a regular JAR deployment. The reason for this is that when you run WildFly in domain mode, deployments are automatically propagated to all servers to which the deployment applies; thus distribution of the driver JAR is one less thing for you to worry about!

Any JDBC 4-compliant driver will automatically be recognized and installed into the system by name and version. A JDBC JAR is identified using the Java service provider mechanism. Such JARs will contain a text a file named `META-INF/services/java.sql.Driver`, which contains the name of the class(es) of the Drivers which exist in that JAR. If your JDBC driver JAR is not JDBC 4-compliant, it can be made deployable in one of a few ways.

Modify the JAR

The most straightforward solution is to simply modify the JAR and add the missing file. You can do this from your command shell by:

1. Change to, or create, an empty temporary directory.
2. Create a `META-INF` subdirectory.
3. Create a `META-INF/services` subdirectory.
4. Create a `META-INF/services/java.sql.Driver` file which contains one line - the fully-qualified class name of the JDBC driver.
5. Use the `jar` command-line tool to update the JAR like this:

```
jar \-uf jdbc-driver.jar META-INF/services/java.sql.Driver
```

For a detailed explanation how to deploy JDBC 4 compliant driver jar, please refer to the chapter "Application Deployment".

Datasource Definitions

The datasource itself is defined within the subsystem *datasources*:

```
<subsystem xmlns="urn:jboss:domain:datasources:4.0">
    <datasources>
        <datasource jndi-name="java:jboss/datasources/ExampleDS" pool-name="ExampleDS">
            <connection-url>jdbc:h2:mem:test;DB_CLOSE_DELAY=-1</connection-url>
            <driver>h2</driver>
            <pool>
                <min-pool-size>10</min-pool-size>
                <max-pool-size>20</max-pool-size>
                <prefill>true</prefill>
            </pool>
            <security>
                <user-name>sa</user-name>
                <password>sa</password>
            </security>
        </datasource>
        <xa-datasource jndi-name="java:jboss/datasources/ExampleXADS" pool-name="ExampleXADS">
           <driver>h2</driver>
           <xa-datasource-property name="URL">jdbc:h2:mem:test</xa-datasource-property>
           <xa-pool>
                <min-pool-size>10</min-pool-size>
                <max-pool-size>20</max-pool-size>
                <prefill>true</prefill>
           </xa-pool>
           <security>
                <user-name>sa</user-name>
                <password>sa</password>
           </security>
        </xa-datasource>
        <drivers>
            <driver name="h2" module="com.h2database.h2">
                <xa-datasource-class>org.h2.jdbcx.JdbcDataSource</xa-datasource-class>
            </driver>
        </drivers>
  </datasources>

</subsystem>
```

(See `standalone/configuration/standalone.xml`)

As you can see the datasource references a driver by it's logical name.

You can easily query the same information through the CLI:

```
[standalone@localhost:9990 /] /subsystem=datasources:read-resource(recursive=true)
{
    "outcome" => "success",
    "result" => {
        "data-source" => {"H2DS" => {
            "connection-url" => "jdbc:h2:mem:test;DB_CLOSE_DELAY=-1",
            "jndi-name" => "java:/H2DS",
            "driver-name" => "h2",
            "pool-name" => "H2DS",
            "use-java-context" => true,
            "enabled" => true,
            "jta" => true,
            "pool-prefill" => true,
            "pool-use-strict-min" => false,
            "user-name" => "sa",
            "password" => "sa",
            "flush-strategy" => "FailingConnectionOnly",
            "background-validation" => false,
            "use-fast-fail" => false,
            "validate-on-match" => false,
            "use-ccm" => true
        }},
        "xa-data-source" => undefined,
        "jdbc-driver" => {"h2" => {
            "driver-name" => "h2",
            "driver-module-name" => "com.h2database.h2",
            "driver-xa-datasource-class-name" => "org.h2.jdbcx.JdbcDataSource"
        }}
    }
}


[standalone@localhost:9990 /] /subsystem=datasources:installed-drivers-list
{
    "outcome" => "success",
    "result" => [{
        "driver-name" => "h2",
        "deployment-name" => undefined,
        "driver-module-name" => "com.h2database.h2",
        "module-slot" => "main",
        "driver-xa-datasource-class-name" => "org.h2.jdbcx.JdbcDataSource",
        "driver-class-name" => "org.h2.Driver",
        "driver-major-version" => 1,
        "driver-minor-version" => 3,
        "jdbc-compliant" => true
    }]
}
```

Using the web console or the CLI greatly simplifies the deployment of JDBC drivers and the creation of datasources.

The CLI offers a set of commands to create and modify datasources:

```
[standalone@localhost:9990 /] data-source --help

SYNOPSIS
  data-source --help [--properties | --commands] |
              (--name=<resource_id> (--<property>=<value>)*) |
              (<command> --name=<resource_id> (--<parameter>=<value>)*)
              [--headers={<operation_header> (;<operation_header>)*}]
DESCRIPTION
  The command is used to manage resources of type /subsystem=datasources/data-source.
[...]


[standalone@localhost:9990 /] xa-data-source --help

SYNOPSIS
  xa-data-source --help [--properties | --commands] |
                 (--name=<resource_id> (--<property>=<value>)*) |
                 (<command> --name=<resource_id> (--<parameter>=<value>)*)
                 [--headers={<operation_header> (;<operation_header>)*}]

DESCRIPTION
  The command is used to manage resources of type /subsystem=datasources/xa-data-source.

RESOURCE DESCRIPTION
  A JDBC XA data-source configuration

[...]
```

Using security domains

Information can be found at https://community.jboss.org/wiki/JBossAS7SecurityDomainModel

Component Reference

The datasource subsystem is provided by the IronJacamar project. For a detailed description of the available configuration properties, please consult the project documentation.

- IronJacamar homepage: http://ironjacamar.org/
- Project Documentation: http://ironjacamar.org/documentation.html
- Schema description: http://www.ironjacamar.org/doc/userguide/1.1/en-US/html_single/index.html#deployingds_descriptor

5.22.19 Deployment Scanner configuration

The deployment scanner is only used in standalone mode. Its job is to monitor a directory for new files and to deploy those files. It can be found in `standalone.xml`:

```
<subsystem xmlns="urn:jboss:domain:deployment-scanner:2.0">
   <deployment-scanner scan-interval="5000"
      relative-to="jboss.server.base.dir" path="deployments" />
</subsystem>
```

You can define more `deployment-scanner` entries to scan for deployments from more locations. The configuration showed will scan the `JBOSS_HOME/standalone/deployments` directory every five seconds. The runtime model is shown below, and uses default values for attributes not specified in the xml:

```
[standalone@localhost:9999 /] /subsystem=deployment-scanner:read-resource(recursive=true)
{
    "outcome" => "success",
    "result" => {"scanner" => {"default" => {
        "auto-deploy-exploded" => false,
        "auto-deploy-zipped" => true,
        "deployment-timeout" => 60L,
        "name" => "default",
        "path" => "deployments",
        "relative-to" => "jboss.server.base.dir",
        "scan-enabled" => true,
        "scan-interval" => 5000
    }}}
}
```

The attributes are

Name	Type	Description
`name`	STRING	The name of the scanner. `default` is used if not specified
`path`	STRING	The actual filesystem path to be scanned. Treated as an absolute path, unless the 'relative-to' attribute is specified, in which case the value is treated as relative to that path.
`relative-to`	STRING	Reference to a filesystem path defined in the "paths" section of the server configuration, or one of the system properties specified on startup. In the example above `jboss.server.base.dir` resolves to `JBOSS_HOME/standalone`
`scan-enabled`	BOOLEAN	If true scanning is enabled
`scan-interval`	INT	Periodic interval, in milliseconds, at which the repository should be scanned for changes. A value of less than 1 indicates the repository should only be scanned at initial startup.
`auto-deploy-zipped`	BOOLEAN	Controls whether zipped deployment content should be automatically deployed by the scanner without requiring the user to add a .dodeploy marker file.
`auto-deploy-exploded`	BOOLEAN	Controls whether exploded deployment content should be automatically deployed by the scanner without requiring the user to add a .dodeploy marker file. Setting this to 'true' is not recommended for anything but basic development scenarios, as there is no way to ensure that deployment will not occur in the middle of changes to the content.
`auto-deploy-xml`	BOOLEAN	Controls whether XML content should be automatically deployed by the scanner without requiring a .dodeploy marker file.
`deployment-timeout`	LONG	Timeout, in seconds, a deployment is allows to execute before being canceled. The default is 60 seconds.

Deployment scanners can be added by modifying `standalone.xml` before starting up the server or they can be added and removed at runtime using the CLI

```
[standalone@localhost:9990 /]
/subsystem=deployment-scanner/scanner=new:add(scan-interval=10000,relative-to="jboss.server.base.d
=> "success"}
[standalone@localhost:9990 /] /subsystem=deployment-scanner/scanner=new:remove
{"outcome" => "success"}
```

You can also change the attributes at runtime, so for example to turn off scanning you can do

```
[standalone@localhost:9990 /]
/subsystem=deployment-scanner/scanner=default:write-attribute(name="scan-enabled",value=false)
{"outcome" => "success"}
[standalone@localhost:9990 /] /subsystem=deployment-scanner:read-resource(recursive=true)
{
    "outcome" => "success",
    "result" => {"scanner" => {"default" => {
        "auto-deploy-exploded" => false,
        "auto-deploy-zipped" => true,
        "deployment-timeout" => 60L,
        "name" => "default",
        "path" => "deployments",
        "relative-to" => "jboss.server.base.dir",
        "scan-enabled" => false,
        "scan-interval" => 5000
    }}}
}
```

5.22.20 EE Subsystem Configuration

Overview

The EE subsystem provides common functionality in the Java EE platform, such as the EE Concurrency Utilities (JSR 236) and `@Resource` injection. The subsystem is also responsible for managing the lifecycle of Java EE application's deployments, that is, `.ear` files.

The EE subsystem configuration may be used to:

- customise the deployment of Java EE applications
- create EE Concurrency Utilities instances
- define the default bindings

The subsystem name is ee and this document covers EE subsystem version `2.0`, which XML namespace within WildFly XML configurations is `urn:jboss:domain:ee:2.0`. The path for the subsystem's XML schema, within WildFly's distribution, is `docs/schema/jboss-as-ee_2_0.xsd`.

Subsystem XML configuration example with all elements and attributes specified:

```
<subsystem xmlns="urn:jboss:domain:ee:2.0" >
    <global-modules>
        <module name="org.jboss.logging"
                slot="main"/>
        <module name="org.apache.log4j"
                annotations="true"
                meta-inf="true"
                services="false" />
    </global-modules>
    <ear-subdeployments-isolated>true</ear-subdeployments-isolated>
```

```
    <spec-descriptor-property-replacement>false</spec-descriptor-property-replacement>
    <jboss-descriptor-property-replacement>false</jboss-descriptor-property-replacement>
    <annotation-property-replacement>false</annotation-property-replacement>
    <concurrent>
        <context-services>
            <context-service
                    name="default"
                    jndi-name="java:jboss/ee/concurrency/context/default"
                    use-transaction-setup-provider="true" />
        </context-services>
        <managed-thread-factories>
            <managed-thread-factory
                    name="default"
                    jndi-name="java:jboss/ee/concurrency/factory/default"
                    context-service="default"
                    priority="1" />
        </managed-thread-factories>
        <managed-executor-services>
            <managed-executor-service
                    name="default"
                    jndi-name="java:jboss/ee/concurrency/executor/default"
                    context-service="default"
                    thread-factory="default"
                    hung-task-threshold="60000"
                    core-threads="5"
                    max-threads="25"
                    keepalive-time="5000"
                    queue-length="1000000"
                    reject-policy="RETRY_ABORT" />
        </managed-executor-services>
        <managed-scheduled-executor-services>
            <managed-scheduled-executor-service
                    name="default"
                    jndi-name="java:jboss/ee/concurrency/scheduler/default"
                    context-service="default"
                    thread-factory="default"
                    hung-task-threshold="60000"
                    core-threads="5"
                    keepalive-time="5000"
                    reject-policy="RETRY_ABORT" />
        </managed-scheduled-executor-services>
    </concurrent>
    <default-bindings
            context-service="java:jboss/ee/concurrency/context/default"
            datasource="java:jboss/datasources/ExampleDS"
            jms-connection-factory="java:jboss/DefaultJMSConnectionFactory"
            managed-executor-service="java:jboss/ee/concurrency/executor/default"
            managed-scheduled-executor-service="java:jboss/ee/concurrency/scheduler/default"
            managed-thread-factory="java:jboss/ee/concurrency/factory/default" />
</subsystem>
```

Java EE Application Deployment

The EE subsystem configuration allows the customisation of the deployment behaviour for Java EE Applications.

Global Modules

Global modules is a set of JBoss Modules that will be added as dependencies to the JBoss Module of every Java EE deployment. Such dependencies allows Java EE deployments to see the classes exported by the global modules.

Each global module is defined through the `module` resource, an example of its XML configuration:

```
<global-modules>
    <module name="org.jboss.logging" slot="main"/>
    <module name="org.apache.log4j" annotations="true" meta-inf="true" services="false" />
  </global-modules>
```

The only mandatory attribute is the JBoss Module `name`, the `slot` attribute defaults to `main`, and both define the JBoss Module ID to reference.

The optional `annotations` attribute, which defaults to `false`, indicates if a pre-computed annotation index should be imported from META-INF/jandex.idx

The optional `services` attribute indicates if any services exposed in META-INF/services should be made available to the deployments class loader, and defaults to `false`.

The optional `meta-inf` attribute, which defaults to `true`, indicates if the Module's `META-INF` path should be available to the deployment's class loader.

EAR Subdeployments Isolation

A flag indicating whether each of the subdeployments within a `.ear` can access classes belonging to another subdeployment within the same `.ear`. The default value is `false`, which allows the subdeployments to see classes belonging to other subdeployments within the `.ear`.

```
<ear-subdeployments-isolated>true</ear-subdeployments-isolated>
```

For example:

```
myapp.ear
|
|--- web.war
|
|--- ejb1.jar
|
|--- ejb2.jar
```

If the `ear-subdeployments-isolated` is set to false, then the classes in `web.war` can access classes belonging to `ejb1.jar` and `ejb2.jar`. Similarly, classes from `ejb1.jar` can access classes from `ejb2.jar` (and vice-versa).

This flag has no effect on the isolated classloader of the `.war` file(s), i.e. irrespective of whether this flag is set to `true` or `false`, the `.war` within a `.ear` will have a isolated classloader, and other subdeployments within that `.ear` will not be able to access classes from that `.war`. This is as per spec.

Property Replacement

The EE subsystem configuration includes flags to configure whether system property replacement will be done on XML descriptors and Java Annotations included in Java EE deployments.

System properties etc are resolved in the security context of the application server itself, not the deployment that contains the file. This means that if you are running with a security manager and enable this property, a deployment can potentially access system properties or environment entries that the security manager would have otherwise prevented.

Spec Descriptor Property Replacement

Flag indicating whether system property replacement will be performed on standard Java EE XML descriptors. If not configured this defaults to `true`, however it is set to `false` in the standard configuration files shipped with WildFly.

```
<spec-descriptor-property-replacement>false</spec-descriptor-property-replacement>
```

JBoss Descriptor Property Replacement

Flag indicating whether system property replacement will be performed on WildFly proprietary XML descriptors, such as `jboss-app.xml`. This defaults to `true`.

```
<jboss-descriptor-property-replacement>false</jboss-descriptor-property-replacement>
```

Annotation Property Replacement

Flag indicating whether system property replacement will be performed on Java annotations. The default value is `false`.

```
<annotation-property-replacement>false</annotation-property-replacement>
```

EE Concurrency Utilities

EE Concurrency Utilities (JSR 236) were introduced with Java EE 7, to ease the task of writing multithreaded Java EE applications. Instances of these utilities are managed by WildFly, and the related configuration provided by the EE subsystem.

Context Services

The Context Service is a concurrency utility which creates contextual proxies from existent objects. WildFly Context Services are also used to propagate the context from a Java EE application invocation thread, to the threads internally used by the other EE Concurrency Utilities. Context Service instances may be created using the subsystem XML configuration:

```
<context-services>
    <context-service
 name="default"
 jndi-name="java:jboss/ee/concurrency/context/default"
 use-transaction-setup-provider="true" />
  </context-services>
```

The `name` attribute is mandatory, and it's value should be a unique name within all Context Services.

The `jndi-name` attribute is also mandatory, and defines where in the JNDI the Context Service should be placed.

The optional `use-trasaction-setup-provider` attribute indicates if the contextual proxies built by the Context Service should suspend transactions in context, when invoking the proxy objects, and its value defaults to true.

Management clients, such as the WildFly CLI, may also be used to configure Context Service instances. An example to `add` and `remove` one named `other`:

```
/subsystem=ee/context-service=other:add(jndi-name=java\:jboss\/ee\/concurrency\/other)
/subsystem=ee/context-service=other:remove
```

Managed Thread Factories

The Managed Thread Factory allows Java EE applications to create new threads. WildFly Managed Thread Factory instances may also, optionally, use a Context Service instance to propagate the Java EE application thread's context to the new threads. Instance creation is done through the EE subsystem, by editing the subsystem XML configuration:

```
<managed-thread-factories>
    <managed-thread-factory
 name="default"
 jndi-name="java:jboss/ee/concurrency/factory/default"
 context-service="default"
 priority="1" />
  </managed-thread-factories>
```

The `name` attribute is mandatory, and it's value should be a unique name within all Managed Thread Factories.

The `jndi-name` attribute is also mandatory, and defines where in the JNDI the Managed Thread Factory should be placed.

The optional `context-service` references an existent Context Service by its `name`. If specified then thread created by the factory will propagate the invocation context, present when creating the thread.

The optional `priority` indicates the priority for new threads created by the factory, and defaults to `5`.

Management clients, such as the WildFly CLI, may also be used to configure Managed Thread Factory instances. An example to `add` and `remove` one named `other`:

```
/subsystem=ee/managed-thread-factory=other:add(jndi-name=java\:jboss\/ee\/factory\/other)
/subsystem=ee/managed-thread-factory=other:remove
```

Managed Executor Services

The Managed Executor Service is the Java EE adaptation of Java SE Executor Service, providing to Java EE applications the functionality of asynchronous task execution. WildFly is responsible to manage the lifecycle of Managed Executor Service instances, which are specified through the EE subsystem XML configuration:

```
<managed-executor-services>
    <managed-executor-service
 name="default"
 jndi-name="java:jboss/ee/concurrency/executor/default"
 context-service="default"
 thread-factory="default"
 hung-task-threshold="60000"
 core-threads="5"
 max-threads="25"
 keepalive-time="5000"
 queue-length="1000000"
 reject-policy="RETRY_ABORT" />
  </managed-executor-services>
```

The `name` attribute is mandatory, and it's value should be a unique name within all Managed Executor Services.

The `jndi-name` attribute is also mandatory, and defines where in the JNDI the Managed Executor Service should be placed.

The optional `context-service` references an existent Context Service by its `name`. If specified then the referenced Context Service will capture the invocation context present when submitting a task to the executor, which will then be used when executing the task.

The optional `thread-factory` references an existent Managed Thread Factory by its `name`, to handle the creation of internal threads. If not specified then a Managed Thread Factory with default configuration will be created and used internally.

The mandatory `core-threads` provides the number of threads to keep in the executor's pool, even if they are idle. A value of `0` means there is no limit.

The optional `queue-length` indicates the number of tasks that can be stored in the input queue. The default value is `0`, which means the queue capacity is unlimited.

The executor’s task queue is based on the values of the attributes `core-threads` and `queue-length`:

- If `queue-length` is `0`, or `queue-length` is `Integer.MAX_VALUE (2147483647)` and `core-threads` is `0`, direct handoff queuing strategy will be used and a synchronous queue will be created.
- If `queue-length` is `Integer.MAX_VALUE` but `core-threads` is not `0`, an unbounded queue will be used.
- For any other valid value for `queue-length`, a bounded queue wil be created.

The optional `hung-task-threshold` defines a threshold value, in milliseconds, to hung a possibly blocked task. A value of `0` will never hung a task, and is the default.

The optional `long-running-tasks` is a hint to optimize the execution of long running tasks, and defaults to `false`.

The optional `max-threads` defines the the maximum number of threads used by the executor, which defaults to Integer.MAX_VALUE (2147483647).

The optional `keepalive-time` defines the time, in milliseconds, that an internal thread may be idle. The attribute default value is `60000`.

The optional reject-policy defines the policy to use when a task is rejected by the executor. The attribute value may be the default `ABORT`, which means an exception should be thrown, or `RETRY_ABORT`, which means the executor will try to submit it once more, before throwing an exception.

Management clients, such as the WildFly CLI, may also be used to configure Managed Executor Service instances. An example to `add` and `remove` one named `other`:

```
/subsystem=ee/managed-executor-service=other:add(jndi-name=java\:jboss\/ee\/executor\/other,
core-threads=2)
/subsystem=ee/managed-executor-service=other:remove
```

Managed Scheduled Executor Services

The Managed Scheduled Executor Service is the Java EE adaptation of Java SE Scheduled Executor Service, providing to Java EE applications the functionality of scheduling task execution. WildFly is responsible to manage the lifecycle of Managed Scheduled Executor Service instances, which are specified through the EE subsystem XML configuration:

```
<managed-scheduled-executor-services>
    <managed-scheduled-executor-service
 name="default"
 jndi-name="java:jboss/ee/concurrency/scheduler/default"
 context-service="default"
 thread-factory="default"
 hung-task-threshold="60000"
 core-threads="5"
 keepalive-time="5000"
 reject-policy="RETRY_ABORT" />
  </managed-scheduled-executor-services>
```

The `name` attribute is mandatory, and it's value should be a unique name within all Managed Scheduled Executor Services.

The `jndi-name` attribute is also mandatory, and defines where in the JNDI the Managed Scheduled Executor Service should be placed.

The optional `context-service` references an existent Context Service by its `name`. If specified then the referenced Context Service will capture the invocation context present when submitting a task to the executor, which will then be used when executing the task.

The optional `thread-factory` references an existent Managed Thread Factory by its `name`, to handle the creation of internal threads. If not specified then a Managed Thread Factory with default configuration will be created and used internally.

The mandatory `core-threads` provides the number of threads to keep in the executor's pool, even if they are idle. A value of `0` means there is no limit.

The optional `hung-task-threshold` defines a threshold value, in milliseconds, to hung a possibly blocked task. A value of `0` will never hung a task, and is the default.

The optional `long-running-tasks` is a hint to optimize the execution of long running tasks, and defaults to `false`.

The optional `keepalive-time` defines the time, in milliseconds, that an internal thread may be idle. The attribute default value is `60000`.

The optional reject-policy defines the policy to use when a task is rejected by the executor. The attribute value may be the default `ABORT`, which means an exception should be thrown, or`RETRY_ABORT`, which means the executor will try to submit it once more, before throwing an exception.

Management clients, such as the WildFly CLI, may also be used to configure Managed Scheduled Executor Service instances. An example to `add` and `remove` one named `other`:

```
/subsystem=ee/managed-scheduled-executor-service=other:add(jndi-name=java\:jboss\/ee\/scheduler\/o
core-threads=2)
/subsystem=ee/managed-scheduled-executor-service=other:remove
```

Default EE Bindings

The Java EE Specification mandates the existence of a default instance for each of the following resources:

- Context Service
- Datasource
- JMS Connection Factory
- Managed Executor Service
- Managed Scheduled Executor Service
- Managed Thread Factory

The EE subsystem looks up the default instances from JNDI, using the names in the default bindings configuration, before placing those in the standard JNDI names, such as `java:comp/DefaultManagedExecutorService`:

```
<default-bindings
 context-service="java:jboss/ee/concurrency/context/default"
 datasource="java:jboss/datasources/ExampleDS"
 jms-connection-factory="java:jboss/DefaultJMSConnectionFactory"
 managed-executor-service="java:jboss/ee/concurrency/executor/default"
 managed-scheduled-executor-service="java:jboss/ee/concurrency/scheduler/default"
 managed-thread-factory="java:jboss/ee/concurrency/factory/default" />
```

The default bindings are optional, if the jndi name for a default binding is not configured then the related resource will not be available to Java EE applications.

Default EE Bindings

The Java EE Specification mandates the existence of a default instance for each of the following resources:

- Context Service
- Datasource
- JMS Connection Factory
- Managed Executor Service
- Managed Scheduled Executor Service
- Managed Thread Factory

The EE subsystem looks up the default instances from JNDI, using the names in the default bindings configuration, before placing those in the standard JNDI names, such as `java:comp/DefaultManagedExecutorService`:

```
<default-bindings
 context-service="java:jboss/ee/concurrency/context/default"
 datasource="java:jboss/datasources/ExampleDS"
 jms-connection-factory="java:jboss/DefaultJMSConnectionFactory"
 managed-executor-service="java:jboss/ee/concurrency/executor/default"
 managed-scheduled-executor-service="java:jboss/ee/concurrency/scheduler/default"
 managed-thread-factory="java:jboss/ee/concurrency/factory/default" />
```

The default bindings are optional, if the jndi name for a default binding is not configured then the related resource will not be available to Java EE applications.

EE Concurrency Utilities

EE Concurrency Utilities (JSR 236) were introduced with Java EE 7, to ease the task of writing multithreaded Java EE applications. Instances of these utilities are managed by WildFly, and the related configuration provided by the EE subsystem.

Context Services

The Context Service is a concurrency utility which creates contextual proxies from existent objects. WildFly Context Services are also used to propagate the context from a Java EE application invocation thread, to the threads internally used by the other EE Concurrency Utilities. Context Service instances may be created using the subsystem XML configuration:

```
<context-services>
    <context-service
 name="default"
 jndi-name="java:jboss/ee/concurrency/context/default"
 use-transaction-setup-provider="true" />
  </context-services>
```

The `name` attribute is mandatory, and it's value should be a unique name within all Context Services.

The `jndi-name` attribute is also mandatory, and defines where in the JNDI the Context Service should be placed.

The optional `use-trasaction-setup-provider` attribute indicates if the contextual proxies built by the Context Service should suspend transactions in context, when invoking the proxy objects, and its value defaults to true.

Management clients, such as the WildFly CLI, may also be used to configure Context Service instances. An example to `add` and `remove` one named `other`:

```
/subsystem=ee/context-service=other:add(jndi-name=java\:jboss\/ee\/concurrency\/other)
/subsystem=ee/context-service=other:remove
```

Managed Thread Factories

The Managed Thread Factory allows Java EE applications to create new threads. WildFly Managed Thread Factory instances may also, optionally, use a Context Service instance to propagate the Java EE application thread's context to the new threads. Instance creation is done through the EE subsystem, by editing the subsystem XML configuration:

```
<managed-thread-factories>
    <managed-thread-factory
 name="default"
 jndi-name="java:jboss/ee/concurrency/factory/default"
 context-service="default"
 priority="1" />
  </managed-thread-factories>
```

The `name` attribute is mandatory, and it's value should be a unique name within all Managed Thread Factories.

The `jndi-name` attribute is also mandatory, and defines where in the JNDI the Managed Thread Factory should be placed.

The optional `context-service` references an existent Context Service by its `name`. If specified then thread created by the factory will propagate the invocation context, present when creating the thread.

The optional `priority` indicates the priority for new threads created by the factory, and defaults to `5`.

Management clients, such as the WildFly CLI, may also be used to configure Managed Thread Factory instances. An example to `add` and `remove` one named `other`:

```
/subsystem=ee/managed-thread-factory=other:add(jndi-name=java\:jboss\/ee\/factory\/other)
/subsystem=ee/managed-thread-factory=other:remove
```

Managed Executor Services

The Managed Executor Service is the Java EE adaptation of Java SE Executor Service, providing to Java EE applications the functionality of asynchronous task execution. WildFly is responsible to manage the lifecycle of Managed Executor Service instances, which are specified through the EE subsystem XML configuration:

```
<managed-executor-services>
    <managed-executor-service
 name="default"
 jndi-name="java:jboss/ee/concurrency/executor/default"
 context-service="default"
 thread-factory="default"
 hung-task-threshold="60000"
 core-threads="5"
 max-threads="25"
 keepalive-time="5000"
 queue-length="1000000"
 reject-policy="RETRY_ABORT" />
  </managed-executor-services>
```

The `name` attribute is mandatory, and it's value should be a unique name within all Managed Executor Services.

The `jndi-name` attribute is also mandatory, and defines where in the JNDI the Managed Executor Service should be placed.

The optional `context-service` references an existent Context Service by its `name`. If specified then the referenced Context Service will capture the invocation context present when submitting a task to the executor, which will then be used when executing the task.

The optional `thread-factory` references an existent Managed Thread Factory by its `name`, to handle the creation of internal threads. If not specified then a Managed Thread Factory with default configuration will be created and used internally.

The mandatory `core-threads` provides the number of threads to keep in the executor's pool, even if they are idle. A value of `0` means there is no limit.

The optional `queue-length` indicates the number of tasks that can be stored in the input queue. The default value is `0`, which means the queue capacity is unlimited.

The executor’s task queue is based on the values of the attributes `core-threads` and `queue-length`:

- If `queue-length` is `0`, or `queue-length` is `Integer.MAX_VALUE (2147483647)` and `core-threads` is `0`, direct handoff queuing strategy will be used and a synchronous queue will be created.
- If `queue-length` is `Integer.MAX_VALUE` but `core-threads` is not `0`, an unbounded queue will be used.
- For any other valid value for `queue-length`, a bounded queue wil be created.

The optional `hung-task-threshold` defines a threshold value, in milliseconds, to hung a possibly blocked task. A value of `0` will never hung a task, and is the default.

The optional `long-running-tasks` is a hint to optimize the execution of long running tasks, and defaults to `false`.

The optional `max-threads` defines the the maximum number of threads used by the executor, which defaults to Integer.MAX_VALUE (2147483647).

The optional `keepalive-time` defines the time, in milliseconds, that an internal thread may be idle. The attribute default value is `60000`.

The optional reject-policy defines the policy to use when a task is rejected by the executor. The attribute value may be the default `ABORT`, which means an exception should be thrown, or `RETRY_ABORT`, which means the executor will try to submit it once more, before throwing an exception.

Management clients, such as the WildFly CLI, may also be used to configure Managed Executor Service instances. An example to `add` and `remove` one named `other`:

```
/subsystem=ee/managed-executor-service=other:add(jndi-name=java\:jboss\/ee\/executor\/other,
core-threads=2)
/subsystem=ee/managed-executor-service=other:remove
```

Managed Scheduled Executor Services

The Managed Scheduled Executor Service is the Java EE adaptation of Java SE Scheduled Executor Service, providing to Java EE applications the functionality of scheduling task execution. WildFly is responsible to manage the lifecycle of Managed Scheduled Executor Service instances, which are specified through the EE subsystem XML configuration:

```
<managed-scheduled-executor-services>
    <managed-scheduled-executor-service
 name="default"
 jndi-name="java:jboss/ee/concurrency/scheduler/default"
 context-service="default"
 thread-factory="default"
 hung-task-threshold="60000"
 core-threads="5"
 keepalive-time="5000"
 reject-policy="RETRY_ABORT" />
  </managed-scheduled-executor-services>
```

The `name` attribute is mandatory, and it's value should be a unique name within all Managed Scheduled Executor Services.

The `jndi-name` attribute is also mandatory, and defines where in the JNDI the Managed Scheduled Executor Service should be placed.

The optional `context-service` references an existent Context Service by its `name`. If specified then the referenced Context Service will capture the invocation context present when submitting a task to the executor, which will then be used when executing the task.

The optional `thread-factory` references an existent Managed Thread Factory by its `name`, to handle the creation of internal threads. If not specified then a Managed Thread Factory with default configuration will be created and used internally.

The mandatory `core-threads` provides the number of threads to keep in the executor's pool, even if they are idle. A value of `0` means there is no limit.

The optional `hung-task-threshold` defines a threshold value, in milliseconds, to hung a possibly blocked task. A value of `0` will never hung a task, and is the default.

The optional `long-running-tasks` is a hint to optimize the execution of long running tasks, and defaults to `false`.

The optional `keepalive-time` defines the time, in milliseconds, that an internal thread may be idle. The attribute default value is `60000`.

The optional reject-policy defines the policy to use when a task is rejected by the executor. The attribute value may be the default `ABORT`, which means an exception should be thrown, or`RETRY_ABORT`, which means the executor will try to submit it once more, before throwing an exception.

Management clients, such as the WildFly CLI, may also be used to configure Managed Scheduled Executor Service instances. An example to `add` and `remove` one named `other`:

```
/subsystem=ee/managed-scheduled-executor-service=other:add(jndi-name=java\:jboss\/ee\/scheduler\/o
core-threads=2)
/subsystem=ee/managed-scheduled-executor-service=other:remove
```

Java EE Application Deployment

The EE subsystem configuration allows the customisation of the deployment behaviour for Java EE Applications.

Global Modules

Global modules is a set of JBoss Modules that will be added as dependencies to the JBoss Module of every Java EE deployment. Such dependencies allows Java EE deployments to see the classes exported by the global modules.

Each global module is defined through the `module` resource, an example of its XML configuration:

```
<global-modules>
    <module name="org.jboss.logging" slot="main"/>
    <module name="org.apache.log4j" annotations="true" meta-inf="true" services="false" />
  </global-modules>
```

The only mandatory attribute is the JBoss Module `name`, the `slot` attribute defaults to `main`, and both define the JBoss Module ID to reference.

The optional `annotations` attribute, which defaults to `false`, indicates if a pre-computed annotation index should be imported from META-INF/jandex.idx

The optional `services` attribute indicates if any services exposed in META-INF/services should be made available to the deployments class loader, and defaults to `false`.

The optional `meta-inf` attribute, which defaults to `true`, indicates if the Module's `META-INF` path should be available to the deployment's class loader.

EAR Subdeployments Isolation

A flag indicating whether each of the subdeployments within a `.ear` can access classes belonging to another subdeployment within the same `.ear`. The default value is `false`, which allows the subdeployments to see classes belonging to other subdeployments within the `.ear`.

```
<ear-subdeployments-isolated>true</ear-subdeployments-isolated>
```

For example:

```
myapp.ear
|
|--- web.war
|
|--- ejb1.jar
|
|--- ejb2.jar
```

If the `ear-subdeployments-isolated` is set to false, then the classes in `web.war` can access classes belonging to `ejb1.jar` and `ejb2.jar`. Similarly, classes from `ejb1.jar` can access classes from `ejb2.jar` (and vice-versa).

This flag has no effect on the isolated classloader of the `.war` file(s), i.e. irrespective of whether this flag is set to `true` or `false`, the `.war` within a `.ear` will have a isolated classloader, and other subdeployments within that `.ear` will not be able to access classes from that `.war`. This is as per spec.

Property Replacement

The EE subsystem configuration includes flags to configure whether system property replacement will be done on XML descriptors and Java Annotations included in Java EE deployments.

System properties etc are resolved in the security context of the application server itself, not the deployment that contains the file. This means that if you are running with a security manager and enable this property, a deployment can potentially access system properties or environment entries that the security manager would have otherwise prevented.

Spec Descriptor Property Replacement

Flag indicating whether system property replacement will be performed on standard Java EE XML descriptors. If not configured this defaults to `true`, however it is set to `false` in the standard configuration files shipped with WildFly.

```
<spec-descriptor-property-replacement>false</spec-descriptor-property-replacement>
```

JBoss Descriptor Property Replacement

Flag indicating whether system property replacement will be performed on WildFly proprietary XML descriptors, such as `jboss-app.xml`. This defaults to `true`.

```
<jboss-descriptor-property-replacement>false</jboss-descriptor-property-replacement>
```

Annotation Property Replacement

Flag indicating whether system property replacement will be performed on Java annotations. The default value is `false`.

```
<annotation-property-replacement>false</annotation-property-replacement>
```

5.22.21 JMX subsystem configuration

The JMX subsystem registers a service with the Remoting endpoint so that remote access to JMX can be obtained over the exposed Remoting connector.

This is switched on by default in standalone mode and accessible over port 9990 but in domain mode is switched off so needs to be enabled - in domain mode the port will be the port of the Remoting connector for the WildFly instance to be monitored.

To use the connector you can access it in the standard way using a `service:jmx` URL:

```
import javax.management.MBeanServerConnection;
import javax.management.remote.JMXConnector;
import javax.management.remote.JMXConnectorFactory;
import javax.management.remote.JMXServiceURL;

public class JMXExample {

    public static void main(String[] args) throws Exception {
        //Get a connection to the WildFly MBean server on localhost
        String host = "localhost";
        int port = 9990;  // management-web port
        String urlString =
            System.getProperty("jmx.service.url","service:jmx:remote+http://" + host + ":" +
port);
        JMXServiceURL serviceURL = new JMXServiceURL(urlString);
        JMXConnector jmxConnector = JMXConnectorFactory.connect(serviceURL, null);
        MBeanServerConnection connection = jmxConnector.getMBeanServerConnection();

        //Invoke on the WildFly MBean server
        int count = connection.getMBeanCount();
        System.out.println(count);
        jmxConnector.close();
    }
}
```

You also need to set your classpath when running the above example. The following script covers Linux. If your environment is much different, paste your script when you have it working.

```
!/bin/bash

# specify your WildFly folder
export YOUR_JBOSS_HOME=~/WildFly

java -classpath $YOUR_JBOSS_HOME/bin/client/jboss-client.jar:./ JMXExample
```

You can also connect using jconsole.

If using jconsole use the `jconsole.sh` and `jconsole.bat` scripts included in the /bin directory of the WildFly distribution as these set the classpath as required to connect over Remoting.

In addition to the standard JVM MBeans, the WildFly MBean server contains the following MBeans:

JMX ObjectName	Description
`jboss.msc:type=container,name=jboss-as`	Exposes management operations on the JBoss Modular Service Container, which is the dependency injection framework at the heart of WildFly. It is useful for debugging dependency problems, for example if you are integrating your own subsystems, as it exposes operations to dump all services and their current states
`jboss.naming:type=JNDIView`	Shows what is bound in JNDI
`jboss.modules:type=ModuleLoader,name=*`	This collection of MBeans exposes management operations on JBoss Modules classloading layer. It is useful for debugging dependency problems arising from missing module dependencies

Audit logging

Audit logging for the JMX MBean server managed by the JMX subsystem. The resource is at `/subsystem=jmx/configuration=audit-log` and its attributes are similar to the ones mentioned for `/core-service=management/access=audit/logger=audit-log` in Audit logging.

Attribute	Description
`enabled`	`true` to enable logging of the JMX operations
`log-boot`	`true` to log the JMX operations when booting the server, `false` otherwise
`log-read-only`	If `true` all operations will be audit logged, if `false` only operations that change the model will be logged

Then which handlers are used to log the management operations are configured as `handler=*` children of the logger. These handlers and their formatters are defined in the global `/core-service=management/access=audit` section mentioned in Audit logging.

JSON Formatter

The same JSON Formatter is used as described in Audit logging. However the records for MBean Server invocations have slightly different fields from those logged for the core management layer.

```
2013-08-29 18:26:29 - {
    "type" : "jmx",
    "r/o" : false,
    "booting" : false,
    "version" : "10.0.0.Final",
    "user" : "$local",
    "domainUUID" : null,
    "access" : "JMX",
    "remote-address" : "127.0.0.1/127.0.0.1",
    "method" : "invoke",
    "sig" : [
        "javax.management.ObjectName",
        "java.lang.String",
        "[Ljava.lang.Object;",
        "[Ljava.lang.String;"
    ],
    "params" : [
        "java.lang:type=Threading",
        "getThreadInfo",
        "[Ljava.lang.Object;@5e6c33c",
        "[Ljava.lang.String;@4b681c69"
    ]
}
```

It includes an optional timestamp and then the following information in the json record

Field name	Description
`type`	This will have the value `jmx` meaning it comes from the jmx subsystem
`r/o`	`true` if the operation has read only impact on the MBean(s)
`booting`	`true` if the operation was executed during the bootup process, `false` if it was executed once the server is up and running
`version`	The version number of the WildFly instance
`user`	The username of the authenticated user.
`domainUUID`	This is not currently populated for JMX operations
`access`	This can have one of the following values: *`NATIVE` - The operation came in through the native management interface, for example the CLI *`HTTP` - The operation came in through the domain HTTP interface, for example the admin console *`JMX` - The operation came in through the JMX subsystem. See JMX for how to configure audit logging for JMX.
`remote-address`	The address of the client executing this operation
`method`	The name of the called MBeanServer method
`sig`	The signature of the called called MBeanServer method
`params`	The actual parameters passed in to the MBeanServer method, a simple `Object.toString()` is called on each parameter.
`error`	If calling the MBeanServer method resulted in an error, this field will be populated with `Throwable.getMessage()`

5.22.22 JSF Configuration

Overview

JSF configuration is handled by the JSF subsystem. The JSF subsystem allows multiple JSF implementations to be installed on the same WildFly server. In particular, any version of Mojarra or MyFaces that implements spec level 2.1 or higher can be installed. For each JSF implementation, a new slot needs to be created under `com.sun.jsf-impl`, `javax.faces.api`, and `org.jboss.as.jsf-injection`. When the JSF subsystem starts up, it scans the module path to find all of the JSF implementations that have been installed. The default JSF implementation that WildFly should use is defined by the `default-jsf-impl-slot` attribute.

Installing a new JSF implementation manually

A new JSF implementation can be manually installed as follows:

Add a module slot for the new JSF implementation JAR

- Create the following directory structure under the WILDFLY_HOME/modules directory:
  WILDFLY_HOME/modules/com/sun/jsf-impl/<JSF_IMPL_NAME>-<JSF_VERSION>

  For example, for Mojarra 2.2.11, the above path would resolve to:
  WILDFLY_HOME/modules/com/sun/jsf-impl/mojarra-2.2.11

- Place the JSF implementation JAR in the <JSF_IMPL_NAME>-<JSF_VERSION> subdirectory. In the same subdirectory, add a `module.xml` file similar to the Mojarra or MyFaces template examples. Change the `resource-root-path` to the name of your JSF implementation JAR and fill in appropriate values for ${`jsf-impl-name`} and ${`jsf-version`}.

Add a module slot for the new JSF API JAR

- Create the following directory structure under the WILDFLY_HOME/modules directory:
  WILDFLY_HOME/modules/javax/faces/api/<JSF_IMPL_NAME>-<JSF_VERSION>

- Place the JSF API JAR in the <JSF_IMPL_NAME>-<JSF_VERSION> subdirectory. In the same subdirectory, add a `module.xml` file similar to the Mojarra or MyFaces template examples. Change the `resource-root-path` to the name of your JSF API JAR and fill in appropriate values for ${`jsf-impl-name`} and ${`jsf-version`}.

Add a module slot for the JSF injection JAR

- Create the following directory structure under the WILDFLY_HOME/modules directory: WILDFLY_HOME/modules/org/jboss/as/jsf-injection/<JSF_IMPL_NAME>-<JSF_VERSION>
- Copy the wildfly-jsf-injection JAR and the weld-core-jsf JAR from WILDFLY_HOME/modules/system/layers/base/org/jboss/as/jsf-injection/main to the <JSF_IMPL_NAME>-<JSF_VERSION> subdirectory.
- In the <JSF_IMPL_NAME>-<JSF_VERSION> subdirectory, add a `module.xml` file similar to the Mojarra or MyFaces template examples and fill in appropriate values for ${`jsf-impl-name`}, ${`jsf-version`}, ${`version.jboss.as`}, and ${`version.weld.core`}. (These last two placeholders depend on the versions of the wildfly-jsf-injection and weld-core-jsf JARs that were copied over in the previous step.)

For MyFaces only - add a module for the commons-digester JAR

- Create the following directory structure under the WILDFLY_HOME/modules directory: WILDFLY_HOME/modules/org/apache/commons/digester/main
- Place the commons-digester JAR in WILDFLY_HOME/modules/org/apache/commons/digester/main. In the `main` subdirectory, add a `module.xml` file similar to this template. Fill in the appropriate value for ${`version.commons-digester`}.

Start the server

After starting the server, the following CLI command can be used to verify that your new JSF implementation has been installed successfully. The new JSF implementation should appear in the output of this command.

```
[standalone@localhost:9990 /] /subsystem=jsf:list-active-jsf-impls()
```

Changing the default JSF implementation

The following CLI command can be used to make a newly installed JSF implementation the default JSF implementation used by WildFly:

```
/subsystem=jsf:write-attribute(name=default-jsf-impl-slot,value=<JSF_IMPL_NAME>-<JSF_VERSION>)
```

A server restart will be required for this change to take effect.

Configuring a JSF app to use a non-default JSF implementation

A JSF app can be configured to use an installed JSF implementation that's not the default implementation by adding a `org.jboss.jbossfaces.JSF_CONFIG_NAME` context parameter to its `web.xml` file. For example, to indicate that a JSF app should use MyFaces 2.2.12 (assuming MyFaces 2.2.12 has been installed on the server), the following context parameter would need to be added:

```
<context-param>
  <param-name>org.jboss.jbossfaces.JSF_CONFIG_NAME</param-name>
  <param-value>myfaces-2.2.12</param-value>
</context-param>
```

If a JSF app does not specify this context parameter, the default JSF implementation will be used for that app.

5.22.23 Logging Configuration

Overview

The overall server logging configuration is represented by the logging subsystem. It consists of four notable parts: `handler` configurations, `logger`, the `root logger` declarations (aka log categories) and logging profiles. Each logger does reference a handler (or set of handlers). Each handler declares the log format and output:

```
<subsystem xmlns="urn:jboss:domain:logging:3.0">
   <console-handler name="CONSOLE" autoflush="true">
       <level name="DEBUG"/>
       <formatter>
           <named-formatter name="COLOR-PATTERN"/>
       </formatter>
   </console-handler>
   <periodic-rotating-file-handler name="FILE" autoflush="true">
       <formatter>
           <named-formatter name="PATTERN"/>
       </formatter>
       <file relative-to="jboss.server.log.dir" path="server.log"/>
       <suffix value=".yyyy-MM-dd"/>
   </periodic-rotating-file-handler>
   <logger category="com.arjuna">
       <level name="WARN"/>
   </logger>
   [...]
   <root-logger>
       <level name="DEBUG"/>
       <handlers>
           <handler name="CONSOLE"/>
           <handler name="FILE"/>
       </handlers>
   </root-logger>
   <formatter name="PATTERN">
       <pattern-formatter pattern="%d{yyyy-MM-dd HH:mm:ss,SSS} %-5p [%c] (%t) %s%e%n"/>
   </formatter>
   <formatter name="COLOR-PATTERN">
       <pattern-formatter pattern="%K{level}%d{HH:mm:ss,SSS} %-5p [%c] (%t) %s%e%n"/>
   </formatter>
</subsystem>
```

Attributes

The root resource contains two notable attributes `add-logging-api-dependencies` and `use-deployment-logging-config`.

logging-api-dependencies

The `add-logging-api-dependencies` controls whether or not the container adds implicit logging API dependencies to your deployments. If set to `true`, the default, all the implicit logging API dependencies are added. If set to `false` the dependencies are not added to your deployments.

deployment-logging-config

The `use-deployment-logging-config` controls whether or not your deployment is scanned for per-deployment logging. If set to `true`, the default, per-deployment logging is enabled. Set to `false` to disable this feature.

deployment Logging

Per-deployment logging allows you to add a logging configuration file to your deployment and have the logging for that deployment configured according to the configuration file. In an EAR the configuration should be in the `META-INF` directory. In a WAR or JAR deployment the configuration file can be in either the `META-INF` or `WEB-INF/classes` directories.

The following configuration files are allowed:

- logging.properties
- jboss-logging.properties
- log4j.properties
- log4j.xml
- jboss-log4j.xml

You can also disable this functionality by changing the `use-deployment-logging-config` attribute to `false`.

Logging Profiles

Logging profiles are like additional logging subsystems. Each logging profile constists of three of the four notable parts listed above: `handler` configurations, `logger` and the `root logger` declarations.

You can assign a logging profile to a deployment via the deployments manifest. Add a `Logging-Profile` entry to the `MANIFEST.MF` file with a value of the logging profile id. For example a logging profile defined on `/subsystem=logging/logging-profile=ejbs` the MANIFEST.MF would look like:

```
Manifest-Version: 1.0
Logging-Profile: ejbs
```

A logging profile can be assigned to any number of deployments. Using a logging profile also allows for runtime changes to the configuration. This is an advantage over the per-deployment logging configuration as the redeploy is not required for logging changes to take affect.

Default Log File Locations

Managed Domain

In a managed domain two types of log files do exist: Controller and server logs. The controller components govern the domain as whole. It's their responsibility to start/stop server instances and execute managed operations throughout the domain. Server logs contain the logging information for a particular server instance. They are co-located with the host the server is running on.

For the sake of simplicity we look at the default setup for managed domain. In this case, both the domain controller components and the servers are located on the same host:

Process	Log File
Host Controller	./domain/log/host-controller.log
Process Controller	./domain/log/process-controller.log
"Server One"	./domain/servers/server-one/log/server.log
"Server Two"	./domain/servers/server-two/log/server.log
"Server Three"	./domain/servers/server-three/log/server.log

Standalone Server

The default log files for a standalone server can be found in the log subdirectory of the distribution:

Process	Log File
Server	./standalone/log/server.log

Filter Expressions

Filter Type	Expression	Description	Parameter(s)	Examples
accept	accept	Accepts all log messages.	None	accept
deny	deny	enies all log messages.	None	deny
not	not(filterExpression)	Accepts a filter as an argument and inverts the returned value.	The expression takes a single filter for it's argument.	not(match("JBAS'
all	all(filterExpressions)	A filter consisting of several filters in a chain. If any filter find the log message to be unloggable, the message will not be logged and subsequent filters will not be checked.	The expression takes a comma delimited list of filters for it's argument.	all(match("JBAS") match("WELD"))
any	any(filterExpressions)	A filter consisting of several filters in a chain. If any filter fins the log message to be loggable, the message will be logged and the subsequent filters will not be checked.	The expression takes a comma delimited list of filters for it's argument.	any(match("JBAS match("WELD"))
levelChange	levelChange(level)	A filter which modifies the log record with a new level.	The expression takes a single string based level for it's argument.	levelChange(WAF

levels	levels(levels)	A filter which includes log messages with a level that is listed in the list of levels.	The expression takes a comma delimited list of string based levels for it's argument.	levels(DEBUG, IN WARN, ERROR)
levelRange	levelRange([minLevel,maxLevel])	A filter which logs records that are within the level range.	The filter expression uses a "[" to indicate a minimum inclusive level and a "]" to indicate a maximum inclusive level. Otherwise use "(" or ")" respectively indicate exclusive. The first argument for the expression is the minimum level allowed, the second argument is the maximum level allowed.	• minimum le be less tha ERROR ar maximum l be greater DEBUG levelRange DEBUG) • minimum le be less tha equal to EF and the ma level must greater tha DEBUG levelRange DEBUG) • minimum le be less tha equal to EF and the ma level must greater or e INFO levelRange INFO]
match	match("pattern")	A regular-expression based filter. The raw unformatted message is used against the pattern.	The expression takes a regular expression for it's argument. match("JBAS\d+")	

substitute	substitute("pattern", "replacement value")	A filter which replaces the first match to the pattern with the replacement value.	The first argument for the expression is the pattern the second argument is the replacement text.	substitute("JBAS"
substituteAll	substituteAll("pattern", "replacement value")	A filter which replaces all matches of the pattern with the replacement value.	The first argument for the expression is the pattern the second argument is the replacement text.	substituteAll("JBA "EAP")

List Log Files and Reading Log Files

Log files can be listed and viewed via management operations. The log files allowed to be viewed are intentionally limited to files that exist in the `jboss.server.log.dir` and are associated with a known file handler. Known file handler types include `file-handler`, `periodic-rotating-file-handler` and `size-rotating-file-handler`. The operations are valid in both standalone and domain modes.

List Log Files

The logging subsystem has a `log-file` resource off the subsystem root resource and off each `logging-profile` resource to list each log file.

CLI command and output

```
[standalone@localhost:9990 /] /subsystem=logging:read-children-names(child-type=log-file)
{
    "outcome" => "success",
    "result" => [
        "server.log",
        "server.log.2014-02-12",
        "server.log.2014-02-13"
    ]
}
```

Read Log File

The `read-log-file` operation is available on each `log-file` resource. This operation has 4 optional parameters.

Name	Description
encoding	the encoding the file should be read in
lines	the number of lines from the file. A value of -1 indicates all lines should be read.
skip	the number of lines to skip before reading.
tail	true to read from the end of the file up or false to read top down.

CLI command and output

```
[standalone@localhost:9990 /] /subsystem=logging/log-file=server.log:read-log-file
{
    "outcome" => "success",
    "result" => [
        "2014-02-14 14:16:48,781 INFO  [org.jboss.as.server.deployment.scanner] (MSC service
thread 1-11) JBAS015012: Started FileSystemDeploymentService for directory
/home/jperkins/servers/wildfly-8.0.0.Final/standalone/deployments",
        "2014-02-14 14:16:48,782 INFO  [org.jboss.as.connector.subsystems.datasources] (MSC
service thread 1-8) JBAS010400: Bound data source [java:jboss/myDs]",
        "2014-02-14 14:16:48,782 INFO  [org.jboss.as.connector.subsystems.datasources] (MSC
service thread 1-15) JBAS010400: Bound data source [java:jboss/datasources/ExampleDS]",
        "2014-02-14 14:16:48,786 INFO  [org.jboss.as.server.deployment] (MSC service thread 1-9)
JBAS015876: Starting deployment of \"simple-servlet.war\" (runtime-name:
\"simple-servlet.war\")",
        "2014-02-14 14:16:48,978 INFO  [org.jboss.ws.common.management] (MSC service thread
1-10) JBWS022052: Starting JBoss Web Services - Stack CXF Server 4.2.3.Final",
        "2014-02-14 14:16:49,160 INFO  [org.wildfly.extension.undertow] (MSC service thread
1-16) JBAS017534: Registered web context: /simple-servlet",
        "2014-02-14 14:16:49,189 INFO  [org.jboss.as.server] (Controller Boot Thread)
JBAS018559: Deployed \"simple-servlet.war\" (runtime-name : \"simple-servlet.war\")",
        "2014-02-14 14:16:49,224 INFO  [org.jboss.as] (Controller Boot Thread) JBAS015961: Http
management interface listening on http://127.0.0.1:9990/management",
        "2014-02-14 14:16:49,224 INFO  [org.jboss.as] (Controller Boot Thread) JBAS015951: Admin
console listening on http://127.0.0.1:9990",
        "2014-02-14 14:16:49,225 INFO  [org.jboss.as] (Controller Boot Thread) JBAS015874:
WildFly 8.0.0.Final \"WildFly\" started in 1906ms - Started 258 of 312 services (90 services are
lazy, passive or on-demand)"
    ]
}
```

FAQ

Why is there a logging.properties file?

You may have noticed that there is a `logging.properties` file in the configuration directory. This is logging configuration is used when the server boots up until the logging subsystem kicks in. If the logging subsystem is not included in your configuration, then this would act as the logging configuration for the entire server.

The `logging.properties` file is overwritten at boot and with each change to the logging subsystem. Any changes made to the file are not persisted. Any changes made to the XML configuration or via management operations will be persisted to the `logging.properties` file and used on the next boot.

Handlers

WIP

This is still a work in progress. Please feel free to edit any mistakes you find .

Overview

Handlers are used to determine what happens with a log message if the logger determines the message is loggable.

There are 6 main handlers provided with WildFly and 1 generic handler;

- async-handler
- console-handler
- file-handler
- periodic-rotating-file-handler
- size-rotating-file-handler
- syslog-handler
- custom-handler

async-handler

An `async-handler` is a handler that asynchronously writes log messages to it's child handlers. This type of handler is generally used for other handlers that take a substantial time to write logged messages.

Attributes

- enabled
- filter-spec
- level
- overflow-action
- queue-length
- subhandlers

console-handler

A `console-handler` is a handler that writes log messages to the console. Generally this writes to `stdout`, but can be set to write to `stderr`.

Attributes

- autoflush
- enabled
- encoding
- filter-spec
- formatter
- level
- named-formatter
- target

file-handler

A `file-handler` is a handler that writes log messages to the specified file.

Attributes

- autoflush
- enabled
- encoding
- filter-spec
- formatter
- level
- named-formatter
- file

periodic-rotating-file-handler

A `periodic-rotating-file-handler` is a handler that writes log messages to the specified file. The file rotates on the date pattern specified in the suffix attribute. The suffix must be a valid pattern recognized by the `java.text.SimpleDateFormat` and must not rotate on seconds or milliseconds.

The rotate happens before the next message is written by the handler.

Attributes

- autoflush
- enabled
- encoding
- filter-spec
- formatter
- level
- named-formatter
- file
- suffix

size-rotating-file-handler

A `size-rotating-file-handler` is a handler that writes log messages to the specified file. The file rotates when the file size is greater than the rotate-size attribute. The rotated file will be kept and the index appended to the name moving previously rotated file indexes up by 1 until the max-backup-index is reached. Once the max-backup-index is reached, the indexed files will be overwritten.

The rotate happens before the next message is written by the handler.

Attributes

- autoflush
- enabled
- encoding
- filter-spec
- formatter
- level
- named-formatter
- file
- max-backup-index
- rotate-size
- rotate-on-boot

syslog-handler

A `syslog-handler` is a handler that writes to a syslog server. The handler support RFC3164 or RFC5424 formats.

Attributes

- port
- app-name
- enabled
- level
- facility
- server-address
- hostname
- syslog-format

The syslog-handler is missing some configuration properties that may be useful in some scenarios like setting a formatter. Use the `org.jboss.logmanager.handlers.SyslogHandler` in module `org.jboss.logmanager` as a custom-handler to exploit these benefits. Additional attributes will be added at some point so this will no longer be necessary.

custom-handler

Attributes

autoflush

Description:	Indicates whether a flush should happen after each write.
Type:	boolean
Default Value:	true
Allowed Values:	true or false

enabled

Description:	If set to true the handler is enabled and functioning as normal, if set to false the handler is ignored when processing log messages.
Type:	boolean
Default Value:	true
Allowed Values:	true or false

encoding

Description:	The character encoding used by this Handler.
Type:	string
Default Value:	null
Allowed Values:	Any valid encoding

file

Description:	An object describing the file the handler should write to.
Type:	object
Default Value:	null
Allowed Values:	An object optionally containing a relative-to property and a path. The path is a required property of the object.

named-formatter

Description:	The name of a defined formatter to be used on the handler.
Type:	string
Default Value:	null
Allowed Values:	**TODO** add link

formatter

Description:	Defines a pattern for a pattern formatter.
Type:	string
Default Value:	%d{HH:mm:ss,SSS} %-5p [%c] (%t) %s%E%n
Allowed Values:	**TODO** add link

filter-spec

Description:	A filter expression value to define a filter.
Type:	string
Default Value:	null
Allowed Values:	See Filter Expression

level

Description:	The log level specifying which message levels will be logged by this logger. Message levels lower than this value will be discarded.
Type:	string
Default Value:	ALL
Allowed Values:	• ALL • FINEST • FINER • TRACE • DEBUG • FINE • CONFIG • INFO • WARN • WARNING • ERROR • SEVERE • FATAL • OFF

backup-index

Description:	The maximum number of rotated files to keep.
Type:	integer
Default Value:	1
Allowed Values:	any integer greater than 0

overflow-action

Description:	Specify what action to take when the overflowing.
Type:	string
Default Value:	BLOCK
Allowed Values:	BLOCK or DISCARD

queue-length

Description:	The queue length to use before flushing writing
Type:	integer
Default Value:	0
Allowed Values:	any positive integer

rotate-on-boot

Description:	Indicates whether or not the file should be rotated each time the file attribute is changed.  If set to `true` will rotate on each boot of the server.
Type:	boolean
Default Value:	false
Allowed Values:	true or false

rotate-size

Description:	The size at which the file should be rotated.
Type:	string
Default Value:	2m
Allowed Values:	Any positive integer with a size type appended to the end. Valid types are b for bytes, k for kilobytes, m for megabytes, g for gigabytes or t for terabytes. Type character is not case sensitive.

subhandlers

Description:	The handlers to associate with the async handler
Type:	list of strings
Default Value:	null
Allowed Values:	An array of valid handler names

suffix

Description:	The pattern used to determine when the file should be rotated.
Type:	string
Default Value:	null
Allowed Values:	Any valid `java.text.SimpleDateFormat` pattern.

target

Description:	Defines the target of the console handler.
Type:	string
Default Value:	System.out
Allowed Values:	System.out or System.err

How To

- How do I add a log category?
- How do I change a log level?
- How do I log my applications messages to their own file?
- How do I use log4j.properties or log4j.xml instead of using the logging subsystem configuration?
- How do I use my own version of log4j?

How do I add a log category?

```
/subsystem=logging/logger=com.your.category:add
```

How do I change a log level?

To change a handlers log level:

```
/subsystem=logging/console-handler=CONSOLE:write-attribute(name=level,value=DEBUG)
```

Changing the level on a log category is the same:

```
/subsystem=logging/logger=com.your.category:write-attribute(name=level,value=ALL)
```

How do I log my applications messages to their own file?

1. Create a file handler. There are 3 different types of file handlers to choose from; `file-handler`, `periodic-rotating-file-handler` and `size-rotating-file-handler`. In this example we'll just use a simple `file-handler`.

```
/subsystem=logging/file-handler=fh:add(level=INFO,
file={"relative-to"=>"jboss.server.log.dir", "path"=>"fh.log"}, append=false,
autoflush=true)
```

2. Now create the log category.

```
/subsystem=logging/logger=org.your.company:add(use-parent-handlers=false,handlers=["fh"])
```

How do I use log4j.properties or log4j.xml instead of using the logging subsystem configuration?

First note that if you choose to use a log4j configuration file, you will no longer be able to make runtime logging changes to your deployments logging configuration.

If that is acceptable you can use per-deployment logging and just include a configuration file in your deployment.

How do I use my own version of log4j?

If you need/want to include your version of log4j then you need to do the following two steps.

1. Disable the adding of the logging dependencies to all your deployments with the add-logging-api-dependencies attribute and disable the use-deployment-logging-config attribute **OR** exclude the logging subsystem in a jboss-deployment-structure.xml.
2. Then need to include a log4j library in your deployment.

This only works for logging in your deployment. Server logs will continue to use the logging subsystem configuration.

Loggers

- Overview
  - Logger Resource
    - filter-spec
    - handlers
    - level
    - use-parent-handlers
  - Root Logger Resource
  - Logger Hierarchy

WIP

This is still a work in progress. Please feel free to edit any mistakes you find  .

Overview

Loggers are used to log messages. A logger is defined by a category generally consisting of a package name or a class name.

A logger is the first step to determining if a messages should be logged or not. If a logger is defined with a level, the level of the message must be greater than the level defined on the logger. The filter is then checked next and the rules of the filter will determine whether or not the messages is said to be loggable.

Logger Resource

A logger resource uses the path `subsystem=logging/logger=$category` where `$category` is the of the logger. For example to a logger named org.wildfly.example would have a resource path of `subsystem=logging/logger=org.wildfly.example`.

A logger as 4 writeable attributes;

- filter-spec
- handlers
- level
- use-parent-handlers

> You may notice that the `category` and `filter` attributes are missing. While `filter` is writable it may be deprecated and removed in the future. Both attributes are still on the resource for legacy reasons.

filter-spec
The `filter-spec` attribute is an expression based string to define filters for the logger.

> Filters on loggers are not inherited.

handlers
The `handlers` attribute is a list of handler names that should be attached to the logger. If the use-parent-handlers attribute is set to `true` and the log messages is determined to be loggable, parent loggers will continue to be processed.
level
The `level` attribute allows the minimum level to allow messages to be logged at for the logger.
parent-handlers
The `use-parent-handlers` attribute is a boolean attribute to determine whether or not parent loggers should also process the log message.

Root Logger Resource

The `root-logger` is similar to a Logger Resource only it has no category and it's name is must be `ROOT`.

Logger Hierarchy

A logger hierarchy is defined by it's category. The category is a `.` (dot) delimited string generally consisting of the package name or a class name. For example the logger `org.wildfly` is the parent logger of `org.wildfly.example`.

5.22.24 Messaging configuration

The JMS server configuration is done through the *messaging-activemq* subsystem. In this chapter we are going outline the frequently used configuration options. For a more detailed explanation please consult the Artemis user guide (See "Component Reference").

Required Extension

The configuration options discussed in this section assume that the the `org.wildfly.extension.messaging-activemq` extension is present in your configuration. This extension is not included in the standard `standalone.xml` and `standalone-ha.xml` configurations included in the WildFly distribution. It is, however, included with the `standalone-full.xml` and `standalone-full-ha.xml` configurations.

You can add the extension to a configuration without it either by adding an `<extension module="org.wildfly.extension.messaging-activemq"/>` element to the xml or by using the following CLI operation:

```
[standalone@localhost:9990 /]/extension=org.wildfly.extension.messaging-activemq:add
```

Connectors

There are three kind of connectors that can be used to connect to WildFly JMS Server

- `in-vm-connector` can be used by a local client (i.e. one running in the same JVM as the server)
- `remote-connector` can be used by a remote client (and uses Netty over TCP for the communication)
- `http-connector` can be used by a remote client (and uses Undertow Web Server to upgrade from a HTTP connection)

JMS Connection Factories

There are three kinds of *basic* JMS `connection-factory` that depends on the type of connectors that is used.

There is also a `pooled-connection-factory` which is special in that it is essentially a configuration facade for *both* the inbound and outbound connectors of the the Artemis JCA Resource Adapter. An MDB can be configured to use a `pooled-connection-factory` (e.g. using `@ResourceAdapter`). In this context, the MDB leverages the *inbound connector* of the Artemis JCA RA. Other kinds of clients can look up the pooled-connection-factory in JNDI (or inject it) and use it to send messages. In this context, such a client would leverage the *outbound connector* of the Artemis JCA RA. A `pooled-connection-factory` is also special because:

- It is only available to local clients, although it can be configured to point to a remote server.
- As the name suggests, it is pooled and therefore provides superior performance to the clients which are able to use it. The pool size can be configured via the `max-pool-size` and `min-pool-size` attributes.
- It should only be used to *send* (i.e. produce) messages when looked up in JNDI or injected.
- It can be configured to use specific security credentials via the `user` and `password` attributes. This is useful if the remote server to which it is pointing is secured.
- Resources acquired from it will be automatically enlisted any on-going JTA transaction. If you want to send a message from an EJB using CMT then this is likely the connection factory you want to use so the send operation will be atomically committed along with the rest of the EJB's transaction operations.

To be clear, the *inbound connector* of the Artemis JCA RA (which is for consuming messages) is only used by MDBs and other JCA-based components. It is not available to traditional clients.

Both a `connection-factory` and a `pooled-connection-factory` reference a `connector` declaration.

A `remote-connector` is associated with a `socket-binding` which tells the client using the `connection-factory` where to connect.

- A `connection-factory` referencing a `remote-connector` is suitable to be used by a *remote* client to send messages to or receive messages from the server (assuming the connection-factory has an appropriately exported `entry`).
- A `pooled-connection-factory` looked up in JNDI or injected which is referencing a `remote-connector` is suitable to be used by a *local* client to send messages to a remote server granted the `socket-binding` references an `outbound-socket-binding` pointing to the remote server in question.
- A `pooled-connection-factory` used by an MDB which is referencing a `remote-connector` is suitable to consume messages from a remote server granted the `socket-binding` references an `outbound-socket-binding` pointing to the remote server in question.

An `in-vm-connector` is associated with a `server-id` which tells the client using the `connection-factory` where to connect (since multiple Artemis servers can run in a single JVM).

- A `connection-factory` referencing an `in-vm-connector` is suitable to be used by a *local* client to either send messages to or receive messages from a local server.
- A `pooled-connection-factory` looked up in JNDI or injected which is referencing an `in-vm-connector` is suitable to be used by a *local* client only to send messages to a local server.
- A `pooled-connection-factory` used by an MDB which is referencing an `in-vm-connector` is suitable only to consume messages from a local server.

A `http-connector` is associated with the `socket-binding` that represents the HTTP socket (by default, named `http`).

- A `connection-factory` referencing a `http-connector` is suitable to be used by a remote client to send messages to or receive messages from the server by connecting to its HTTP port before upgrading to the messaging protocol.
- A `pooled-connection-factory` referencing a `http-connector` is suitable to be used by a local client to send messages to a remote server granted the `socket-binding` references an `outbound-socket-binding` pointing to the remote server in question.
- A `pooled-connection-factory` used by an MDB which is referencing a `http-connector` is suitable only to consume messages from a remote server granted the `socket-binding` references an `outbound-socket-binding` pointing to the remote server in question.

The `entry` declaration of a `connection-factory` or a `pooled-connection-factory` specifies the JNDI name under which the factory will be exposed. Only JNDI names bound in the `"java:jboss/exported"` namespace are available to remote clients. If a `connection-factory` has an entry bound in the `"java:jboss/exported"` namespace a remote client would look-up the `connection-factory` using the text *after* `"java:jboss/exported"`. For example, the "`RemoteConnectionFactory`" is bound by default to `"java:jboss/exported/jms/RemoteConnectionFactory"` which means a remote client would look-up this `connection-factory` using "`jms/RemoteConnectionFactory`". A `pooled-connection-factory` should *not* have any `entry` bound in the "`java:jboss/exported`" namespace because a `pooled-connection-factory` is not suitable for remote clients.

Since JMS 2.0, a default JMS connection factory is accessible to EE application under the JNDI name `java:comp/DefaultJMSConnectionFactory.` WildFly messaging subsystem defines a `pooled-connection-factory` that is used to provide this default connection factory. Any parameter change on this `pooled-connection-factory` will be take into account by any EE application looking the default JMS provider under the JNDI name `java:comp/DefaultJMSConnectionFactory.`

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:1.0">
    <server name="default">
        [...]
        <http-connector name="http-connector"
                        socket-binding="http"
                        endpoint="http-acceptor" />
        <http-connector name="http-connector-throughput"
                        socket-binding="http"
                        endpoint="http-acceptor-throughput">
            <param name="batch-delay"
                   value="50"/>
        </http-connector>
        <in-vm-connector name="in-vm"
                         server-id="0"/>
      [...]
      <connection-factory name="InVmConnectionFactory"
                          connectors="in-vm"
                          entries="java:/ConnectionFactory" />
      <pooled-connection-factory name="activemq-ra"
                          transaction="xa"
                          connectors="in-vm"
                          entries="java:/JmsXA java:jboss/DefaultJMSConnectionFactory"/>
      [...]
   </server>
</subsystem>
```

(See standalone/configuration/standalone-full.xml)

JMS Queues and Topics

JMS queues and topics are sub resources of the messaging-actively subsystem. One can define either a `jms-queue` or `jms-topic`. Each destination *must* be given a `name` and contain at least one entry in its `entries` element (separated by whitespace).

Each entry refers to a JNDI name of the queue or topic. Keep in mind that any `jms-queue` or `jms-topic` which needs to be accessed by a remote client needs to have an entry in the "java:jboss/exported" namespace. As with connection factories, if a `jms-queue` or or `jms-topic` has an entry bound in the "java:jboss/exported" namespace a remote client would look it up using the text *after* `"java:jboss/exported`". For example, the following `jms-queue` "testQueue" is bound to "java:jboss/exported/jms/queue/test" which means a remote client would look-up this {{kms-queue} using "jms/queue/test". A local client could look it up using "java:jboss/exported/jms/queue/test", "java:jms/queue/test", or more simply "jms/queue/test":

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:1.0">
    <server name="default">
    [...]
    <jms-queue name="testQueue"
               entries="jms/queue/test java:jboss/exported/jms/queue/test" />
    <jms-topic name="testTopic"
               entries="jms/topic/test java:jboss/exported/jms/topic/test" />
</subsystem>
```

(See standalone/configuration/standalone-full.xml)

JMS endpoints can easily be created through the CLI:

```
[standalone@localhost:9990 /] jms-queue add --queue-address=myQueue --entries=queues/myQueue
```

```
[standalone@localhost:9990 /]
/subsystem=messaging-activemq/server=default/jms-queue=myQueue:read-resource
{
    "outcome" => "success",
    "result" => {
        "durable" => true,
        "entries" => ["queues/myQueue"],
        "selector" => undefined
    }
}
```

A number of additional commands to maintain the JMS subsystem are available as well:

```
[standalone@localhost:9990 /] jms-queue --help --commands
add
...
remove
To read the description of a specific command execute 'jms-queue command_name --help'.
```

Dead Letter & Redelivery

Some of the settings are applied against an address wild card instead of a specific messaging destination. The dead letter queue and redelivery settings belong into this group:

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:1.0">
   <server name="default">
      [...]
      <address-setting name="#"
                       dead-letter-address="jms.queue.DLQ"
                       expiry-address="jms.queue.ExpiryQueue"
                       [...] />
```

(See standalone/configuration/standalone-full.xml)

Security Settings for Artemis addresses and JMS destinations

Security constraints are matched against an address wildcard, similar to the DLQ and redelivery settings.

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:1.0">
   <server name="default">
      [...]
      <security-setting name="#">
          <role name="guest"
                send="true"
                consume="true"
                create-non-durable-queue="true"
                delete-non-durable-queue="true"/>
```

(See standalone/configuration/standalone-full.xml)

Security Domain for Users

By default, Artemis will use the "`other`" JAAS security domain. This domain is used to authenticate users making connections to Artemis and then they are authorized to perform specific functions based on their role(s) and the `security-settings` described above. This domain can be changed by using the `security-domain`, e.g.:

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:1.0">
   <server name="default">
       <security domain="mySecurityDomain" />
      [...]
```

Using the Elytron Subsystem

You can also use the elytron subsystem to secure the messaging-activemq subsystem.

To use an Elytron security domain:

1. Undefine the legacy security domain.

```
/subsystem=messaging-activemq/server=default:undefine-attribute(name=security-domain)
```

2. Set an Elytron security domain.

```
/subsystem=messaging-activemq/server=default:write-attribute(name=elytron-domain,
value=myElytronSecurityDomain)
```

You can only define either `security-domain` or `elytron-domain`, but you cannot have both defined at the same time. If neither is defined, WildFly will use the `security-domain` default value of `other`, which maps to the `other` legacy security domain.

Cluster Authentication

If the Artemis server is configured to be clustered, it will use the `cluster` 's `user` and `password` attributes to connect to other Artemis nodes in the cluster.

If you do not change the default value of <cluster-password>, Artemis will fail to authenticate with the error:

```
HQ224018: Failed to create session: HornetQExceptionerrorType=CLUSTER_SECURITY_EXCEPTION
message=HQ119099: Unable to authenticate cluster user: HORNETQ.CLUSTER.ADMIN.USER
```

To prevent this error, you must specify a value for `<cluster-password>`. It is possible to encrypt this value by following this guide.

Alternatively, you can use the system property jboss.messaging.cluster.password to specify the cluster password from the command line.

Deployment of -jms.xml files

Starting with WildFly 8, you have the ability to deploy a -jms.xml file defining JMS destinations, e.g.:

```
<?xml version="1.0" encoding="UTF-8"?>
<messaging-deployment xmlns="urn:jboss:messaging-activemq-deployment:1.0">
   <server name="default">
      <jms-destinations>
         <jms-queue name="sample">
            <entry name="jms/queue/sample"/>
            <entry name="java:jboss/exported/jms/queue/sample"/>
         </jms-queue>
      </jms-destinations>
   </server>
</messaging-deployment>
```

This feature **is primarily intended for development** as destinations deployed this way can not be managed with any of the provided management tools (e.g. console, CLI, etc).

JMS Bridge

The function of a JMS bridge is to consume messages from a source JMS destination, and send them to a target JMS destination. Typically either the source or the target destinations are on different servers.
The bridge can also be used to bridge messages from other non Artemis JMS servers, as long as they are JMS 1.1 compliant.

The JMS Bridge is provided by the Artemis project. For a detailed description of the available configuration properties, please consult the project documentation.

Modules for other messaging brokers

Source and target JMS resources (destination and connection factories) are looked up using JNDI.
If either the source or the target resources are managed by another messaging server than WildFly, the required client classes must be bundled in a module. The name of the module must then be declared when the JMS Bridge is configured.

The use of a JMS bridges with any messaging provider will require to create a module containing the jar of this provider.

Let's suppose we want to use an hypothetical messaging provider named AcmeMQ. We want to bridge messages coming from a source AcmeMQ destination to a target destination on the local WildFly messaging server. To lookup AcmeMQ resources from JNDI, 2 jars are required, acmemq-1.2.3.jar, mylogapi-0.0.1.jar (please note these jars do not exist, this is just for the example purpose). We must *not* include a JMS jar since it will be provided by a WildFly module directly.

To use these resources in a JMS bridge, we must bundle them in a WildFly module:

in JBOSS_HOME/modules, we create the layout:

```
modules/
`-- org
    `-- acmemq
        `-- main
            |-- acmemq-1.2.3.jar
            |-- mylogapi-0.0.1.jar
            `-- module.xml
```

We define the module in `module.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<module xmlns="urn:jboss:module:1.1" name="org.acmemq">
    <properties>
        <property name="jboss.api" value="private"/>
    </properties>


    <resources>
        <!-- insert resources required to connect to the source or target   -->
        <!-- messaging brokers if it not another WildFly instance           -->
        <resource-root path="acmemq-1.2.3.jar" />
        <resource-root path="mylogapi-0.0.1.jar" />
    </resources>


    <dependencies>
       <!-- add the dependencies required by JMS Bridge code               -->
       <module name="javax.api" />
       <module name="javax.jms.api" />
       <module name="javax.transaction.api"/>
       <module name="org.jboss.remote-naming"/>
       <!-- we depend on org.apache.activemq.artemis module since we will send messages to  -->
       <!-- the Artemis server embedded in the local WildFly instance       -->
       <module name="org.apache.activemq.artemis" />
    </dependencies>
</module>
```

Configuration

A JMS bridge is defined inside a `jms-bridge` section of the `messaging-activemq` subsystem in the XML configuration files.

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:1.0">
   <jms-bridge name="myBridge" module="org.acmemq">
      <source connection-factory="ConnectionFactory"
              destination="sourceQ"
              user="user1"
              password="pwd1"
              quality-of-service="AT_MOST_ONCE"
              failure-retry-interval="500"
              max-retries="1"
              max-batch-size="500"
              max-batch-time="500"
              add-messageID-in-header="true">
         <source-context>
            <property name="java.naming.factory.initial"
                      value="org.acmemq.jndi.AcmeMQInitialContextFactory"/>
            <property name="java.naming.provider.url"
                      value="tcp://127.0.0.1:9292"/>
         </source-context>
      </source>
      <target connection-factory"/jms/invmTargetCF"
              destination="/jms/targetQ" />
      </target>
   </jms-bridge>
</subsystem>
```

The `source` and `target` sections contain the name of the JMS resource (`connection-factory` and `destination`) that will be looked up in JNDI.
It optionally defines the `user` and `password` credentials. If they are set, they will be passed as arguments when creating the JMS connection from the looked up ConnectionFactory.
It is also possible to define JNDI context properties in the `source-context` and `target-context` sections. If these sections are absent, the JMS resources will be looked up in the local WildFly instance (as it is the case in the `target` section in the example above).

Management commands

A JMS Bridge can also be managed using the WildFly command line interface:

```
[standalone@localhost:9990 /] /subsystem=messaging/jms-bridge=myBridge/:add(module="org.acmemq",
\
      source-destination="sourceQ",
\
      source-connection-factory="ConnectionFactory",
\
      source-user="user1",
\
      source-password="pwd1",
\
      source-context={"java.naming.factory.initial" =>
"org.acmemq.jndi.AcmeMQInitialContextFactory", \
                     "java.naming.provider.url" => "tcp://127.0.0.1:9292"},
\
      target-destination="/jms/targetQ",
\
      target-connection-factory="/jms/invmTargetCF",
\
      quality-of-service=AT_MOST_ONCE,
\
      failure-retry-interval=500,
\
      max-retries=1,
\
      max-batch-size=500,
\
      max-batch-time=500,
\
      add-messageID-in-header=true)
{"outcome" => "success"}
```

You can also see the complete JMS Bridge resource description from the CLI:

```
[standalone@localhost:9990 /] /subsystem=messaging/jms-bridge=*/:read-resource-description
{
    "outcome" => "success",
    "result" => [{
        "address" => [
            ("subsystem" => "messaging"),
            ("jms-bridge" => "*")
        ],
        "outcome" => "success",
        "result" => {
            "description" => "A JMS bridge instance.",
            "attributes" => {
                ...
        }
    }]
}
```

Component Reference

The messaging-activemq subsystem is provided by the Artemis project. For a detailed description of the available configuration properties, please consult the project documentation.

- Artemis Homepage: http://activemq.apache.org/artemis/
- Artemis User Documentation: http://activemq.apache.org/artemis/docs.html

Connect a pooled-connection-factory to a Remote Artemis Server

The `messaging-activemq` subsystem allows to configure a `pooled-connection-factory` resource to let a local client deployed in WildFly connect to a remote Artemis server.

The configuration of such a `pooled-connection-factory` is done in 3 steps:

1. create an outbound-socket-binding pointing to the remote messaging server:

```
/socket-binding-group=standard-sockets/remote-destination-outbound-socket-binding=remote-artemis:a
host>, port=61616)
```

2. create a remote-connector referencing the outbound-socket-binding created at step (1).

```
/subsystem=messaging-activemq/server=default/remote-connector=remote-artemis:add(socket-binding=re
```

3. create a pooled-connection-factory referencing the remote-connector created at step (2).

```
/subsystem=messaging-activemq/server=default/pooled-connection-factory=remote-artemis:add(connecto
entries=[java:/jms/remoteCF])
```

Configuration of a MDB using a pooled-connection-factory

When a `pooled-connection-factory` is configured to connect to a remote Artemis, it is possible to configure Message-Driven Beans (MDB) to have them consume messages from this remote server.

The MDB must be annotated with the `@ResourceAdapter` annotation using the **name** of the `pooled-connection-factory resource`

```
import org.jboss.ejb3.annotation.ResourceAdapter;

@ResourceAdapter("remote-artemis")
@MessageDriven(name = "MyMDB", activationConfig = {
    ...
})
public class MyMDB implements MessageListener {
    public void onMessage(Message message) {
       ...
    }
}
```

If the MDB needs to produce messages to the remote server, it must inject the `pooled-connection-factory` by looking it up in JNDI using one of its `entries`.

```
@Inject
@JMSConnectionFactory("java:/jms/remoteCF")
private JMSContext context;
```

Configuration of the destination

A MDB must also specify which destination it will consume messages from.

The standard way is to define a `destinationLookup` activation config property that corresponds to a JNDI lookup on the local server.
When the MDB is consuming from a remote Artemis server, there may not have such a JNDI binding in the local WildFly.
It is possible to use the naming subsystem to configure external context federation to have local JNDI bindings delegating to external bindings.

However there is a simpler solution to configure the destination when using the Artemis Resource Adapter. Instead of using JDNI to lookup the JMS Destination resource, you can just specify the **name** of the destination (as configured in the remote Artemis server) using the `destination` activation config property and set the `useJNDI` activation config property to false to let the Artemis Resource Adapter create automatically the JMS destination without requiring any JNDI lookup.

```
@ResourceAdapter("remote-artemis")
@MessageDriven(name = "MyMDB", activationConfig = {
    @ActivationConfigProperty(propertyName = "useJNDI",         propertyValue = "false"),
    @ActivationConfigProperty(propertyName = "destination",     propertyValue = "myQueue"),
    @ActivationConfigProperty(propertyName = "destinationType", propertyValue =
"javax.jms.Queue"),
    @ActivationConfigProperty(propertyName = "acknowledgeMode", propertyValue =
"Auto-acknowledge")
})
public class MyMDB implements MessageListener {
    ...
}
```

These properties configure the MDB to consume messages from the JMS Queue named `myQueue` hosted on the remote Artemis server.
In most cases, such a MDB does not need to lookup other destinations to process the consumed messages and it can use the JMSReplyTo destination if it is defined on the message.
If the MDB needs any other JMS destinations defined on the remote server, it must use client-side JNDI by following the Artemis documentation or configure external configuration context in the naming subsystem (which allows to inject the JMS resources using the `@Resource` annotation).

Backward & Forward Compatibility

WildFly 10 supports both backwards and forwards compatibility with legacy versions that were using HornetQ as their messaging brokers (such as JBoss AS7, WildFly 8 & 9).
These two compatibility modes are provided by the ActiveMQ Artemis project that supports the HornetQ's CORE protocol:

- backward compatibility: WildFly 10 JMS clients (using Artemis) can connect to legacy app server (running HornetQ)
- forward compatibility: legacy JMS clients (using HornetQ) can connect to WildFly 10 app server (running Artemis).

Forward Compatibility

Forward compatibility requires no code change in legacy JMS clients. It is provided by WildFly 10 messaging-activemq subsystem and its resources.

- `legacy-connection-factory` is a subresource of the `messaging-activemq`'s `server` and can be used to store in JNDI a HornetQ-based JMS ConnectionFactory.

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:1.0">
    <server name="default">
        ...
        <legacy-connection-factory name="legacyConnectionFactory-discovery"
                                   entries="java:jboss/exported/jms/RemoteConnectionFactory"
                                   ... />
    </server>
</subsystem>
```

- Legacy HornetQ-based JMS destinations can also be configured by providing a `legacy-entries` attribute to the `jms-queue` and `jms-topic` resource.

```
<jms-queue name="myQueue"
               entries="java:jboss/exported/jms/myQueue-new"
               legacy-entries="java:jboss/exported/jms/myQueue" />
    <jms-topic name="testTopic"
               entries="java:jboss/exported/jms/myTopic-new"
               legacy-entries="java:jboss/exported/jms/myTopic" />
```

The `legacy-entries` must be used by legacy JMS clients (using HornetQ) while the regular `entries` are for WildFly 10 JMS clients (using Artemis).

The legacy JMS client will then lookup this `legacy` JMS resources to communicate with WildFly 10.
To avoid any code change in the legacy JMS clients, the legacy JNDI entries must match the lookup expected by the legacy JMS client.

Migration

During migration, the legacy `messaging` subsystem will create `legacy-connection-factory` resource and add `legacy-entries` to the `jms-queue` and `jms-topic` resource if the boolean attribute `add-legacy-entries` is set to `true` for its `migrate` operation. If that is the case, the legacy entries in the migrated `messaging-activemq` subsystem will correspond to the entries specified in the legacy `messaging` subsystem and the regular entries will be created with a `-new` suffix.
If `add-legacy-entries` is set to `false` during migration, no legacy resources will be created in the `messaging-activemq` subsystem and legacy JMS clients will not be able to communicate with WildFly 10 servers.

Backward Compatibility

Backward compatibility requires no configuration change in the legacy server.
WildFly 10 JMS clients do not look up resources on the legacy server but uses client-side JNDI to create their JMS resources. Artemis JMS client can then uses these resources to communicate with the legacy server using the HornetQ CORE protocol.

Artemis supports Client-side JNDI to create JMS resources (`ConnectionFactory` and `Destination`).

For example, if a WidFly 10 JMS clients wants to communicate with a legacy server using a JMS queue named `myQueue`, it must use the following properties to configure its JNDI `InitialContext`:

```
java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory
connectionFactory.jms/ConnectionFactory=tcp://<legacy server address>:5445? \

protocolManagerFactoryStr=org.apache.activemq.artemis.core.protocol.hornetq.client.HornetQClientPr
```

It can then use the `jms/ConnectionFactory` name to create the JMS ConnectionFactory and `jms/myQueue` to create the JMS Queue.
Note that the property `protocolManagerFactoryStr=org.apache.activemq.artemis.core.protocol.hornetq.clie` is mandatory when specifying the URL of the legacy connection factory so that the Artemis JMS client can communicate with the HornetQ broker in the legacy server.

AIO - NIO for messaging journal

Apache ActiveMQ Artemis (like HornetQ beforehand) ships with a **high performance journal**. Since Apache ActiveMQ Artemis handles its own persistence, rather than relying on a database or other 3rd party persistence engine it is very highly optimised for the specific messaging use cases. The majority of the journal is written in Java, however we abstract out the interaction with the actual file system to allow different pluggable implementations.

Apache ActiveMQ Artemis ships with two implementations:

- **Java NIO.**

The first implementation uses standard Java NIO to interface with the file system. This provides extremely good performance and runs on any platform where there's a Java 6+ runtime.

- **Linux Asynchronous IO**

The second implementation uses a thin native code wrapper to talk to the Linux asynchronous IO library (AIO). With AIO, Apache ActiveMQ Artemis will be called back when the data has made it to disk, allowing us to avoid explicit syncs altogether and simply send back confirmation of completion when AIO informs us that the data has been persisted.

Using AIO will typically provide even better performance than using Java NIO.

The AIO journal is only available when running Linux kernel 2.6 or later and after having installed libaio (if it's not already installed).

Also, please note that AIO will only work with the following file systems: ext2, ext3, ext4, jfs, xfs. With other file systems, e.g. NFS it may appear to work, but it will fall back to a slower synchronous behaviour. Don't put the journal on a NFS share!

One point that should be added is that AIO doesn't work well with encrypted partitions, thus you have to move to NIO on those.

What are the symptoms of an AIO isssue ?

AIO issue on WildFly 10

If you see the following exception in your WildFly log file / console

```
[org.apache.activemq.artemis.core.server] (ServerService Thread Pool -- 64) AMQ222010: Critical
IO Error, shutting down the server.
file=AIOSequentialFile:/home/wildfly/wildfly-10.0.0.Final/standalone/data/activemq/journal/activem
message=Cannot open file:The Argument is invalid: java.io.IOException: Cannot open file:The
Argument is invalid
 at org.apache.activemq.artemis.jlibaio.LibaioContext.open(Native Method)
```

that means that AIO isn't working properly on your system.

To use NIO instead execute the following commnd using jboss-cli :

```
/subsystem=messaging-activemq/server=default:write-attribute(name=journal-type, value=NIO)
```

You need to reload or restart your server and you should see the following trace in your server console :

```
INFO  [org.apache.activemq.artemis.core.server] (ServerService Thread Pool -- 64) AMQ221013:
Using NIO Journal
```

AIO issue on WildFly 9

```
[org.hornetq.core.server] (ServerService Thread Pool -- 64) HQ222010: Critical IO Error,
shutting down the server.
file=AIOSequentialFile:/home/wildfly/wildfly-9.0.2.Final/standalone/data/messagingjournal/hornetq-
message=Can't open file: HornetQException[errorType=NATIVE_ERROR_CANT_OPEN_CLOSE_FILE
message=Can't open file]
 at org.hornetq.core.libaio.Native.init(Native Method)
```

that means that AIO isn't working properly on your system.

To use NIO instead execute the following commnd using jboss-cli :

```
/subsystem=messaging/hornetq-server=default:write-attribute(name=journal-type,value=NIO)
```

You need to reload or restart your server and you see the following trace in your server console :

```
INFO  [org.hornetq.core.server] (ServerService Thread Pool -- 64) HQ221013: Using NIO Journal
```

JDBC Store for Messaging Journal

The Artemis server that are integrated to WildFly can be configured to use a JDBC store for its messaging journal instead of its file-based journal.
The `server resource` of the messaging-activemq subsystem needs to configure its `journal-datasource` attribute to be able to use JDBC store. If this attribute is not defined, the regular file-base journal will be used for the Artemis server.
This attribute value must correspond to a data source defined in the datasource subsystem.

For example, if the datasources subsystem defines an `ExampleDS` data source at `/subsystem=datasources/data-source=ExampleDS`, the Artemis server can use it for its JDBC store with the operation:

```
/subsystem=messaging-activemq/server=default:write-attribute(name=journal-datasource,
value=ExampleDS)
```

Artemis JDBC store uses SQL commands to create the tables used to persist its information.
These SQL commands may differ depending on the type of database. The SQL commands used by the JDBC store are located in the file at:

```
$JBOSS_HOME/modules/system/layers/base/org/wildfly/extension/messaging-activemq/main/database/jour
```

By default, "vanilla" SQL commands are used to communicate with the database. However some databases requires specific commands to create table, update content, etc.

The journal-sql.properties can also specify these provider-specific commands. You can customize them by adding a suffix to the vanilla SQL properties.

The suffix is determined based on information from the JDBC driver and the connection metadata. If the type of database is not supported by the code, you can specify it explicitly with the journal-database property on the server resource.

Artemis uses different JDBC tables to store its bindings information, the persistent messages and the large messages (paging is not supported yet).

The name of these tables can be configured with the `journal-bindings-table`, `journal-messages-table`, `journal-page-store-table, and journal-large-messages-table`.

Reference

- Artemis JDBC Persistence - http://activemq.apache.org/artemis/docs/1.5.0/persistence.html#configuring-jdbc-persistence

5.22.25 Naming Subsystem Configuration

Overview

The Naming subsystem provides the JNDI implementation on WildFly, and its configuration allows to:

- bind entries in global JNDI namespaces
- turn off/on the remote JNDI interface

The subsystem name is naming and this document covers Naming subsystem version `2.0`, which XML namespace within WildFly XML configurations is `urn:jboss:domain:naming:2.0`. The path for the subsystem's XML schema, within WildFly's distribution, is `docs/schema/jboss-as-naming_2_0.xsd`.

Subsystem XML configuration example with all elements and attributes specified:

```
<subsystem xmlns="urn:jboss:domain:naming:2.0">
    <bindings>
        <simple name="java:global/a" value="100" type="int" />
        <simple name="java:global/jboss.org/docs/url" value="https://docs.jboss.org"
type="java.net.URL" />
        <object-factory name="java:global/foo/bar/factory" module="org.foo.bar"
class="org.foo.bar.ObjectFactory" />
        <external-context name="java:global/federation/ldap/example"
class="javax.naming.directory.InitialDirContext" cache="true">
            <environment>
                <property name="java.naming.factory.initial"
value="com.sun.jndi.ldap.LdapCtxFactory" />
                <property name="java.naming.provider.url" value="ldap://ldap.example.com:389" />
                <property name="java.naming.security.authentication" value="simple" />
                <property name="java.naming.security.principal" value="uid=admin,ou=system" />
                <property name="java.naming.security.credentials" value="secret" />
            </environment>
        </external-context>
        <lookup name="java:global/c" lookup="java:global/b" />
    </bindings>
    <remote-naming/>
</subsystem>
```

Global Bindings Configuration

The Naming subsystem configuration allows binding entries into the following global JNDI namespaces:

- `java:global`
- `java:jboss`
- `java:`

If WildFly is to be used as a Java EE application server, then it's recommended to opt for `java:global`, since it is a standard (i.e. portable) namespace.

Four different types of bindings are supported:

- Simple
- Object Factory
- External Context
- Lookup

In the subsystem's XML configuration, global bindings are configured through the `<bindings />` XML element, as an example:

```
<bindings>
    <simple name="java:global/a" value="100" type="int" />
    <object-factory name="java:global/foo/bar/factory" module="org.foo.bar"
class="org.foo.bar.ObjectFactory" />
    <external-context name="java:global/federation/ldap/example"
class="javax.naming.directory.InitialDirContext" cache="true">
        <environment>
            <property name="java.naming.factory.initial"
value="com.sun.jndi.ldap.LdapCtxFactory" />
            <property name="java.naming.provider.url" value="ldap://ldap.example.com:389" />
            <property name="java.naming.security.authentication" value="simple" />
            <property name="java.naming.security.principal" value="uid=admin,ou=system" />
            <property name="java.naming.security.credentials" value="secret" />
        </environment>
    </external-context>
    <lookup name="java:global/c" lookup="java:global/b" />
</bindings>
```

Simple Bindings

A simple binding is a primitive or java.net.URL entry, and it is defined through the `simple` XML element. An example of its XML configuration:

```
<simple name="java:global/a" value="100" type="int" />
```

The `name` attribute is mandatory and specifies the target JNDI name for the entry.

The `value` attribute is mandatory and defines the entry's value.

The optional `type` attribute, which defaults to `java.lang.String`, specifies the type of the entry's value. Besides java.lang.String, allowed types are all the primitive types and their corresponding object wrapper classes, such as int or java.lang.Integer, and java.net.URL.

Management clients, such as the WildFly CLI, may be used to configure simple bindings. An example to `add` and `remove` the one in the XML example above:

```
/subsystem=naming/binding=java\:global\/a:add(binding-type=simple, type=int, value=100)
/subsystem=naming/binding=java\:global\/a:remove
```

Object Factories

The Naming subsystem configuration allows the binding of `javax.naming.spi.ObjectFactory` entries, through the `object-factory` XML element, for instance:

```
<object-factory name="java:global/foo/bar/factory" module="org.foo.bar"
class="org.foo.bar.ObjectFactory">
    <environment>
        <property name="p1" value="v1" />
        <property name="p2" value="v2" />
    </environment>
</object-factory>
```

The `name` attribute is mandatory and specifies the target JNDI name for the entry.

The `class` attribute is mandatory and defines the object factory's Java type.

The `module` attribute is mandatory and specifies the JBoss Module ID where the object factory Java class may be loaded from.

The optional `environment` child element may be used to provide a custom environment to the object factory.

Management clients, such as the WildFly CLI, may be used to configure object factory bindings. An example to `add` and `remove` the one in the XML example above:

```
/subsystem=naming/binding=java\:global\/foo\/bar\/factory:add(binding-type=object-factory,
module=org.foo.bar, class=org.foo.bar.ObjectFactory, environment=[p1=v1, p2=v2])
/subsystem=naming/binding=java\:global\/foo\/bar\/factory:remove
```

External Context Federation

Federation of external JNDI contexts, such as a LDAP context, are achieved by adding External Context bindings to the global bindings configuration, through the `external-context` XML element. An example of its XML configuration:

```
<external-context name="java:global/federation/ldap/example"
class="javax.naming.directory.InitialDirContext" cache="true">
    <environment>
        <property name="java.naming.factory.initial" value="com.sun.jndi.ldap.LdapCtxFactory" />
        <property name="java.naming.provider.url" value="ldap://ldap.example.com:389" />
        <property name="java.naming.security.authentication" value="simple" />
        <property name="java.naming.security.principal" value="uid=admin,ou=system" />
        <property name="java.naming.security.credentials" value="secret" />
    </environment>
</external-context>
```

The `name` attribute is mandatory and specifies the target JNDI name for the entry.

The `class` attribute is mandatory and indicates the Java initial naming context type used to create the federated context. Note that such type must have a constructor with a single environment map argument.

The optional `module` attribute specifies the JBoss Module ID where any classes required by the external JNDI context may be loaded from.

The optional `cache` attribute, which value defaults to `false`, indicates if the external context instance should be cached.

The optional `environment` child element may be used to provide the custom environment needed to lookup the external context.

Management clients, such as the WildFly CLI, may be used to configure external context bindings. An example to `add` and `remove` the one in the XML example above:

```
/subsystem=naming/binding=java\:global\/federation\/ldap\/example:add(binding-type=external-contex
cache=true, class=javax.naming.directory.InitialDirContext,
environment=[java.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory,
java.naming.provider.url=ldap\:\/\/ldap.example.com\:389,
java.naming.security.authentication=simple,
java.naming.security.principal=uid\=admin\,ou\=system, java.naming.security.credentials=
secret])

/subsystem=naming/binding=java\:global\/federation\/ldap\/example:remove
```

Some JNDI providers may fail when their resources are looked up if they do not implement properly the lookup(Name) method. Their errors would look like:

```
11:31:49,047 ERROR org.jboss.resource.adapter.jms.inflow.JmsActivation (default-threads
-1)  javax.naming.InvalidNameException: Only support CompoundName names
    at com.tibco.tibjms.naming.TibjmsContext.lookup(TibjmsContext.java:504)
    at javax.naming.InitialContext.lookup(InitialContext.java:421)
```

To work around their shortcomings, the `org.jboss.as.naming.lookup.by.string` property can be specified in the external-context's environment to use instead the lookup(String) method (with a performance degradation):

```
<property name="org.jboss.as.naming.lookup.by.string" value="true"/>
```

Binding Aliases

The Naming subsystem configuration allows the binding of existent entries into additional names, i.e. aliases. Binding aliases are specified through the `lookup` XML element. An example of its XML configuration:

```
<lookup name="java:global/c" lookup="java:global/b" />
```

The `name` attribute is mandatory and specifies the target JNDI name for the entry.

The `lookup` attribute is mandatory and indicates the source JNDI name. It can chain lookups on external contexts. For example, having an external context bounded to *java:global/federation/ldap/example*, searching can be done there by setting `lookup` attribute to *java:global/federation/ldap/example/subfolder*.

Management clients, such as the WildFly CLI, may be used to configure binding aliases. An example to `add` and `remove` the one in the XML example above:

```
/subsystem=naming/binding=java\:global\/c:add(binding-type=lookup, lookup=java\:global\/b)
/subsystem=naming/binding=java\:global\/c:remove
```

Remote JNDI Configuration

The Naming subsystem configuration may be used to (de)activate the remote JNDI interface, which allows clients to lookup entries present in a remote WildFly instance.

Only entries within the `java:jboss/exported` context are accessible over remote JNDI.

In the subsystem's XML configuration, remote JNDI access bindings are configured through the `<remote-naming />` XML element:

```
<remote-naming />
```

Management clients, such as the WildFly CLI, may be used to add/remove the remote JNDI interface. An example to `add` and `remove` the one in the XML example above:

```
/subsystem=naming/service=remote-naming:add
/subsystem=naming/service=remote-naming:remove
```

Global Bindings Configuration

The Naming subsystem configuration allows binding entries into the following global JNDI namespaces:

- `java:global`
- `java:jboss`
- `java:`

If WildFly is to be used as a Java EE application server, then it's recommended to opt for `java:global`, since it is a standard (i.e. portable) namespace.

Four different types of bindings are supported:

- Simple
- Object Factory
- External Context
- Lookup

In the subsystem's XML configuration, global bindings are configured through the `<bindings />` XML element, as an example:

```
<bindings>
    <simple name="java:global/a" value="100" type="int" />
    <object-factory name="java:global/foo/bar/factory" module="org.foo.bar"
class="org.foo.bar.ObjectFactory" />
    <external-context name="java:global/federation/ldap/example"
class="javax.naming.directory.InitialDirContext" cache="true">
        <environment>
            <property name="java.naming.factory.initial"
value="com.sun.jndi.ldap.LdapCtxFactory" />
            <property name="java.naming.provider.url" value="ldap://ldap.example.com:389" />
            <property name="java.naming.security.authentication" value="simple" />
            <property name="java.naming.security.principal" value="uid=admin,ou=system" />
            <property name="java.naming.security.credentials" value="secret" />
        </environment>
    </external-context>
    <lookup name="java:global/c" lookup="java:global/b" />
</bindings>
```

Simple Bindings

A simple binding is a primitive or java.net.URL entry, and it is defined through the `simple` XML element. An example of its XML configuration:

```
<simple name="java:global/a" value="100" type="int" />
```

The `name` attribute is mandatory and specifies the target JNDI name for the entry.

The `value` attribute is mandatory and defines the entry's value.

The optional `type` attribute, which defaults to `java.lang.String`, specifies the type of the entry's value. Besides java.lang.String, allowed types are all the primitive types and their corresponding object wrapper classes, such as int or java.lang.Integer, and java.net.URL.

Management clients, such as the WildFly CLI, may be used to configure simple bindings. An example to `add` and `remove` the one in the XML example above:

```
/subsystem=naming/binding=java\:global\/a:add(binding-type=simple, type=int, value=100)
/subsystem=naming/binding=java\:global\/a:remove
```

Object Factories

The Naming subsystem configuration allows the binding of `javax.naming.spi.ObjectFactory` entries, through the `object-factory` XML element, for instance:

```
<object-factory name="java:global/foo/bar/factory" module="org.foo.bar"
class="org.foo.bar.ObjectFactory">
    <environment>
        <property name="p1" value="v1" />
        <property name="p2" value="v2" />
    </environment>
</object-factory>
```

The `name` attribute is mandatory and specifies the target JNDI name for the entry.

The `class` attribute is mandatory and defines the object factory's Java type.

The `module` attribute is mandatory and specifies the JBoss Module ID where the object factory Java class may be loaded from.

The optional `environment` child element may be used to provide a custom environment to the object factory.

Management clients, such as the WildFly CLI, may be used to configure object factory bindings. An example to `add` and `remove` the one in the XML example above:

```
/subsystem=naming/binding=java\:global\/foo\/bar\/factory:add(binding-type=object-factory,
module=org.foo.bar, class=org.foo.bar.ObjectFactory, environment=[p1=v1, p2=v2])
/subsystem=naming/binding=java\:global\/foo\/bar\/factory:remove
```

External Context Federation

Federation of external JNDI contexts, such as a LDAP context, are achieved by adding External Context bindings to the global bindings configuration, through the `external-context` XML element. An example of its XML configuration:

```
<external-context name="java:global/federation/ldap/example"
class="javax.naming.directory.InitialDirContext" cache="true">
    <environment>
        <property name="java.naming.factory.initial" value="com.sun.jndi.ldap.LdapCtxFactory" />
        <property name="java.naming.provider.url" value="ldap://ldap.example.com:389" />
        <property name="java.naming.security.authentication" value="simple" />
        <property name="java.naming.security.principal" value="uid=admin,ou=system" />
        <property name="java.naming.security.credentials" value="secret" />
    </environment>
</external-context>
```

The `name` attribute is mandatory and specifies the target JNDI name for the entry.

The `class` attribute is mandatory and indicates the Java initial naming context type used to create the federated context. Note that such type must have a constructor with a single environment map argument.

The optional `module` attribute specifies the JBoss Module ID where any classes required by the external JNDI context may be loaded from.

The optional `cache` attribute, which value defaults to `false`, indicates if the external context instance should be cached.

The optional `environment` child element may be used to provide the custom environment needed to lookup the external context.

Management clients, such as the WildFly CLI, may be used to configure external context bindings. An example to `add` and `remove` the one in the XML example above:

```
/subsystem=naming/binding=java\:global\/federation\/ldap\/example:add(binding-type=external-contex
cache=true, class=javax.naming.directory.InitialDirContext,
environment=[java.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory,
java.naming.provider.url=ldap\://\/ldap.example.com\:389,
java.naming.security.authentication=simple,
java.naming.security.principal=uid\=admin\,ou\=system, java.naming.security.credentials=
secret])

/subsystem=naming/binding=java\:global\/federation\/ldap\/example:remove
```

Some JNDI providers may fail when their resources are looked up if they do not implement properly the lookup(Name) method. Their errors would look like:

```
11:31:49,047 ERROR org.jboss.resource.adapter.jms.inflow.JmsActivation (default-threads
-1)  javax.naming.InvalidNameException: Only support CompoundName names
    at com.tibco.tibjms.naming.TibjmsContext.lookup(TibjmsContext.java:504)
    at javax.naming.InitialContext.lookup(InitialContext.java:421)
```

To work around their shortcomings, the `org.jboss.as.naming.lookup.by.string` property can be specified in the external-context's environment to use instead the lookup(String) method (with a performance degradation):

```
<property name="org.jboss.as.naming.lookup.by.string" value="true"/>
```

Binding Aliases

The Naming subsystem configuration allows the binding of existent entries into additional names, i.e. aliases. Binding aliases are specified through the `lookup` XML element. An example of its XML configuration:

```
<lookup name="java:global/c" lookup="java:global/b" />
```

The `name` attribute is mandatory and specifies the target JNDI name for the entry.

The `lookup` attribute is mandatory and indicates the source JNDI name. It can chain lookups on external contexts. For example, having an external context bounded to *java:global/federation/ldap/example*, searching can be done there by setting `lookup` attribute to *java:global/federation/ldap/example/subfolder*.

Management clients, such as the WildFly CLI, may be used to configure binding aliases. An example to `add` and `remove` the one in the XML example above:

```
/subsystem=naming/binding=java\:global\/c:add(binding-type=lookup, lookup=java\:global\/b)
/subsystem=naming/binding=java\:global\/c:remove
```

Remote JNDI Configuration

The Naming subsystem configuration may be used to (de)activate the remote JNDI interface, which allows clients to lookup entries present in a remote WildFly instance.

Only entries within the `java:jboss/exported` context are accessible over remote JNDI.

In the subsystem's XML configuration, remote JNDI access bindings are configured through the `<remote-naming />` XML element:

```
<remote-naming />
```

Management clients, such as the WildFly CLI, may be used to add/remove the remote JNDI interface. An example to `add` and `remove` the one in the XML example above:

```
/subsystem=naming/service=remote-naming:add
/subsystem=naming/service=remote-naming:remove
```

5.22.26 Resource adapters

Resource adapters are configured through the *resource-adapters* subsystem. Declaring a new resource adapter consists of two separate steps: You would need to deploy the .rar archive and define a resource adapter entry in the subsystem.

Resource Adapter Definitions

The resource adapter itself is defined within the subsystem *resource-adapters*:

```
<subsystem xmlns="urn:jboss:domain:resource-adapters:1.0">
    <resource-adapters>
       <resource-adapter>
          <archive>eis.rar</archive>
          <!-- Resource adapter level config-property -->
          <config-property name="Server">localhost</config-property>
          <config-property name="Port">19000</config-property>
          <transaction-support>XATransaction</transaction-support>
          <connection-definitions>
             <connection-definition class-name="com.acme.eis.ra.EISManagedConnectionFactory"
                                    jndi-name="java:/eis/AcmeConnectionFactory"
                                    pool-name="AcmeConnectionFactory">
                <!-- Managed connection factory level config-property -->
                <config-property name="Name">Acme Inc</config-property>
                <pool>
                   <min-pool-size>10</min-pool-size>
                   <max-pool-size>100</max-pool-size>
                </pool>
                <security>
                   <application/>
                </security>
             </connection-definition>
          </connection-definitions>
          <admin-objects>
             <admin-object class-name="com.acme.eis.ra.EISAdminObjectImpl"
                           jndi-name="java:/eis/AcmeAdminObject">
                <config-property name="Threshold">10</config-property>
             </admin-object>
          </admin-objects>
       </resource-adapter>
    </resource-adapters>
</subsystem>
```

Note, that only JNDI bindings under java:/ or java:jboss/ are supported.

(See `standalone/configuration/standalone.xml`)

Using security domains

Information about using security domains can be found at
https://community.jboss.org/wiki/JBossAS7SecurityDomainModel

Automatic activation of resource adapter archives

A resource adapter archive can be automatically activated with a configuration by including an META-INF/ironjacamar.xml in the archive.

The schema can be found at http://docs.jboss.org/ironjacamar/schema/ironjacamar_1_0.xsd

Component Reference

The resource adapter subsystem is provided by the IronJacamar project. For a detailed description of the available configuration properties, please consult the project documentation.

- IronJacamar homepage: http://www.jboss.org/ironjacamar
- Project Documentation: http://www.jboss.org/ironjacamar/docs
- Schema description: http://docs.jboss.org/ironjacamar/userguide/1.0/en-US/html/deployment.html#deployingra_descriptor

5.22.27 Security subsystem configuration

The security subsystem is the subsystem that brings the security services provided by PicketBox to the WildFly 8 server instances.

If you are looking to secure the management interfaces for the management of the domain then you should read the Securing the Management Interfaces chapter as the management interfaces themselves are not run within a WildFly process so use a custom configuration.

Structure of the Security Subsystem

When deploying applications to WildFly most of the time it is likely that you would be deploying a web application or EJBs and just require a security domain to be defined with login modules to verify the users identity, this chapter aims to provide additional detail regarding the architecture and capability of the security subsystem however if you are just looking to define a security domain and leave the rest to the container please jump to the security-domains section.

The security subsystem operates by using a security context associated with the current request, this security context then makes available to the relevant container a number of capabilities from the configured security domain, the capabilities exposed are an authentication manager, an authorization manager, an audit manager and a mapping manager.

Authentication Manager

The authentication manager is the component that performs the actual authentication taking the declared users identity and their credential so that the login context for the security domain can be used to 'login' the user using the configured login module or modules.

Authorization Manager

The authorization manager is a component which can be obtained by the container from the current security context to either obtain information about a users roles or to perform an authorization check against a resource for the currently authenticated user.

Audit Manager

The audit manager from the security context is the component that can be used to log audit events in relation to the security domain.

Mapping Manager

The mapping manager can be used to assign additional principals, credentials, roles or attributes to the authenticated subject.

Security Subsystem Configuration

By default a lot of defaults have already been selected for the security subsystem and unless there is a specific implementation detail you need to change, these defaults should not require modification. This chapter describes all of the possible configuration attributes for completeness but do keep in mind that not all will need to be changed.

The security subsystem is enabled by default by the addition of the following extension: -

<extension module="org.jboss.as.security"/>

The namespace used for the configuration of the security subsystem is urn:jboss:domain:security:1.0, the configuration is defined within the <subsystem> element from this namespace.

The <subsystem> element can optionally contain the following child elements.

- security-management
- subject-factory
- security-domains
- security-properties

security-management

This element is used to override some of the high level implementation details of the PicketBox implementation if you have a need to change some of this behaviour.

The element can have any or the following attributes set, all of which are optional.

authentication-manager-class-name	Specifies the AuthenticationManager implementation class name to use.
deep-copy-subject-mode	Sets the copy mode of subjects done by the security managers to be deep copies that makes copies of the subject principals and credentials if they are cloneable. It should be set to true if subject include mutable content that can be corrupted when multiple threads have the same identity and cache flushes/logout clearing the subject in one thread results in subject references affecting other threads. Default value is "false".
default-callback-handler-class-name	Specifies a global class name for the CallbackHandler implementation to be used with login modules.
authorization-manager-class-name	Attribute specifies the AuthorizationManager implementation class name to use.
audit-manager-class-name	Specifies the AuditManager implementation class name to use.
identity-trust-manager-class-name	Specifies the IdentityTrustManager implementation class name to use.
mapping-manager-class-name	Specifies the MappingManager implementation class name to use.

subject-factory

The subject factory is responsible for creating subject instances, this also makes use of the authentication manager to actually verify the caller. It is used mainly by JCA components to establish a subject. It is not likely this would need to be overridden but if it is required the "subject-factory-class-name" attribute can be specified on the subject-factory element.

security-domains

This portion of the configuration is where the bulk of the security subsystem configuration will actually take place for most administrators, the security domains contain the configuration which is specific to a deployment.

The security-domains element can contain numerous <security-domain> definitions, a security-domain can have the following attributes set:

name	The unique name of this security domain.
~~extends~~	Although version 1.0 of the security subsystem schema contained an 'extends' attribute, security domain inheritance is not supported and this attribute should not be used.
cache-type	The type of authentication cache to use with this domain. If this attribute is removed no cache will be used. Allowed values are "default" or "infinispan"

The following elements can then be set within the security-domain to configure the domain behaviour.

authentication

The authentication element is used to hold the list of login modules that will be used for authentication when this domain is used, the structure of the login-module element is:

```
<login-module code="..." flag="..." module="...">
  <module-option name="..." value="..."/>
</login-module>
```

The code attribute is used to specify the implementing class of the login module which can either be the full class name or one of the abbreviated names from the following list:

Code	Classname
Client	`org.jboss.security.ClientLoginModule`
Certificate	`org.jboss.security.auth.spi.BaseCertLoginModule`
CertificateUsers	`org.jboss.security.auth.spi.BaseCertLoginModule`
CertificateRoles	`org.jboss.security.auth.spi.CertRolesLoginModule`
Database	`org.jboss.security.auth.spi.DatabaseServerLoginModule`
DatabaseCertificate	`org.jboss.security.auth.spi.DatabaseCertLoginModule`
DatabaseUsers	`org.jboss.security.auth.spi.DatabaseServerLoginModule`
Identity	`org.jboss.security.auth.spi.IdentityLoginModule`
Ldap	`org.jboss.security.auth.spi.LdapLoginModule`
LdapExtended	`org.jboss.security.auth.spi.LdapExtLoginModule`
RoleMapping	`org.jboss.security.auth.spi.RoleMappingLoginModule`
RunAs	`org.jboss.security.auth.spi.RunAsLoginModule`
Simple	`org.jboss.security.auth.spi.SimpleServerLoginModule`
ConfiguredIdentity	`org.picketbox.datasource.security.ConfiguredIdentityLoginModule`
SecureIdentity	`org.picketbox.datasource.security.SecureIdentityLoginModule`
PropertiesUsers	`org.jboss.security.auth.spi.PropertiesUsersLoginModule`
SimpleUsers	`org.jboss.security.auth.spi.SimpleUsersLoginModule`
LdapUsers	`org.jboss.security.auth.spi.LdapUsersLoginModule`
Kerberos	`com.sun.security.auth.module.Krb5LoginModule`
SPNEGOUsers	`org.jboss.security.negotiation.spnego.SPNEGOLoginModule`
AdvancedLdap	`org.jboss.security.negotiation.AdvancedLdapLoginModule`
AdvancedADLdap	`org.jboss.security.negotiation.AdvancedADLoginModule`
UsersRoles	`org.jboss.security.auth.spi.UsersRolesLoginModule`

The module attribute specifies the name of the JBoss Modules module from which the class specified by the code attribute should be loaded. Specifying it is not necessary if one of the abbreviated names in the above list is used.

The flag attribute is used to specify the JAAS flag for this module and should be one of required, requisite, sufficient, or optional.

The module-option element can be repeated zero or more times to specify the module options as required for the login module being configured. It requires the name and value attributes.

See Authentication Modules for further details on the various modules listed above.

authentication-jaspi

The authentication-jaspi is used to configure a Java Authentication SPI (JASPI) provider as the authentication mechanism. A security domain can have either a <authentication> or a <authentication-jaspi> element, but not both. We set up JASPI by configuring one or more login modules inside the login-module-stack element and setting up an authentication module. Here is the structure of the authentication-jaspi element:

```
<login-module-stack name="...">
  <login-module code="..." flag="..." module="...">
    <module-option name="..." value="..."/>
  </login-module>
</login-module-stack>
<auth-module code="..." login-module-stack-ref="...">
  <module-option name="..." value="..."/>
</auth-module>
```

The login-module-stack-ref attribute value must be the name of the login-module-stack element to be used. The sub-element login-module is configured just like in the authentication part

authorization

Authorization in the AS container is normally done with RBAC (role based access control) but there are situations where a more fine grained authorization policy is required. The authorization element allows definition of different authorization modules to used, such that authorization can be checked with JACC (Java Authorization Contract for Containers) or XACML (eXtensible Access Control Markup Language). The structure of the authorization element is:

```
<policy-module code="..." flag="..." module="...">
  <module-option name="..." value="..."/>
</policy-module>
```

The code attribute is used to specify the implementing class of the policy module which can either be the full class name or one of the abbreviated names from the following list:

Code	Classname
DenyAll	`org.jboss.security.authorization.modules.AllDenyAuthorizationModule`
PermitAll	`org.jboss.security.authorization.modules.AllPermitAuthorizationModule`
Delegating	`org.jboss.security.authorization.modules.DelegatingAuthorizationModule`
Web	`org.jboss.security.authorization.modules.WebAuthorizationModule`
JACC	`org.jboss.security.authorization.modules.JACCAuthorizationModule`
XACML	`org.jboss.security.authorization.modules.XACMLAuthorizationModule`

The module attribute specifies the name of the JBoss Modules module from which the class specified by the code attribute should be loaded. Specifying it is not necessary if one of the abbreviated names in the above list is used.

The flag attribute is used to specify the JAAS flag for this module and should be one of required, requisite, sufficient, or optional.

The module-option element can be repeated zero or more times to specify the module options as required for the login module being configured. It requires the name and value attributes.

mapping

The mapping element defines additional mapping of principals, credentials, roles and attributes for the subject. The structure of the mapping element is:

```
<mapping-module type="..."code="..." module="...">
  <module-option name="..." value="..."/>
</mapping-module>
```

The type attribute reflects the type of mapping of the provider and should be one of principal, credential, role or attribute. By default "role" is the type used if the attribute is not set.

The code attribute is used to specify the implementing class of the login module which can either be the full class name or one of the abbreviated names from the following list:

Code	Classname
PropertiesRoles	`org.jboss.security.mapping.providers.role.PropertiesRolesMappingP`
SimpleRoles	`org.jboss.security.mapping.providers.role.SimpleRolesMappingProvi`
DeploymentRoles	`org.jboss.security.mapping.providers.DeploymentRolesMappingProvid`
DatabaseRoles	`org.jboss.security.mapping.providers.role.DatabaseRolesMappingPro`
LdapRoles	`org.jboss.security.mapping.providers.role.LdapRolesMappingProvide`

The module attribute specifies the name of the JBoss Modules module from which the class specified by the code attribute should be loaded. Specifying it is not necessary if one of the abbreviated names in the above list is used.

The module-option element can be repeated zero or more times to specify the module options as required for the login module being configured. It requires the name and value attributes.

audit

The audit element can be used to define a custom audit provider. The default implementation used is `org.jboss.security.audit.providers.LogAuditProvider`. The structure of the audit element is:

```
<provider-module code="..." module="...">
  <module-option name="..." value="..."/>
</provider-module>
```

The code attribute is used to specify the implementing class of the provider module.

The module attribute specifies the name of the JBoss Modules module from which the class specified by the code attribute should be loaded.

The module-option element can be repeated zero or more times to specify the module options as required for the login module being configured. It requires the name and value attributes.

jsse

The jsse element defines configuration for keystores and trustores that can be used for SSL context configuration or for certificate storing/retrieving.

The set of attributes (all of them optional) of this element are:

keystore-password	Password of the keystore
keystore-type	Type of the keystore. By default it's "JKS"
keystore-url	`URL` where the keystore file can be found
keystore-provider	`Provider` of the keystore. The default JDK provider for the keystore type is used if this attribute is null
keystore-provider-argument	String that can be passed as the argument of the keystore `Provider` constructor
key-manager-factory-algorithm	Algorithm of the `KeyManagerFactory`. The default JDK algorithm of the key manager factory is used if this attribute is null
key-manager-factory-provider	`Provider` of the `KeyManagerFactory`. The default JDK provider for the key manager factory algorithm is used if this attribute is null
truststore-password	Password of the truststore
truststore-type	Type of the truststore. By deafult it's "JKS"
truststore-url	`URL` where the truststore file can be found
truststore-provider	`Provider` of the truststore. The default JDK provider for the truststore type is used if this attribute is null
truststore-provider-argument	String that can be passed as the argument of the truststore `Provider` constructor
trust-manager-factory-algorithm	Algorithm of the `TrustManagerFactory`. The default JDK algorithm of the trust manager factory is used if this attribute is null
trust-manager-factory-provider	`Provider` of the `TrustManagerFactory`. The default JDK provider for the trust manager factory algorithm is used if this attribute is null
client-alias	Alias of the keystore to be used when creating client side SSL sockets
server-alias	Alias of the keystore to be used when creating server side SSL sockets
service-auth-token	Validation token to enable third party services to retrieve a keystore `Key`. This is typically used to retrieve a private key for signing purposes
client-auth	Flag to indicate if the server side SSL socket should require client authentication. Default is "false"
cipher-suites	Comma separated list of cipher suites to be used by a `SSLContext`
protocols	Comma separated list of SSL protocols to be used by a `SSLContext`

The optional additional-properties element can be used to include other options. The structure of the jsse element is:

```
<jsse keystore-url="..." keystore-password="..." keystore-type="..." keystore-provider="..."
keystore-provider-argument="..." key-manager-factory-algorithm="..."
key-manager-factory-provider="..." truststore-url="..." truststore-password="..."
truststore-type="..." truststore-provider="..." truststore-provider-argument="..."
trust-manager-factory-algorithm="..." trust-manager-factory-provider="..." client-alias="..."
server-alias="..." service-auth-token="..." client-auth="..." cipher-suites="..."
protocols="...">
  <additional-properties>x=y
    a=b
  </additional-properties>
</jsse>
```

security-properties

This element is used to specify additional properties as required by the security subsystem, properties are specified in the following format:

```
<security-properties>
  <property name="..." value="..."/>
</security-properties>
```

The property element can be repeated as required for as many properties need to be defined.

Each property specified is set on the `java.security.Security` class.

Authentication Modules

In this section we will describe each login module's options available.

Client

This login module is designed to establish caller identity and credentials when WildFly is acting a client. It should never be used as part of a security domain used for actual server authentication.

Options	Usage	Description
multi-threaded	optional	Set to **true** if each thread has its own principal and credential storage. Set to **false** to indicate that all threads in the VM share the same identity and credential. Default is **false**
restore-login-identity	optional	Set to **true** if the identity and credential seen at the start of the login() method should be restored after the logout() method is invoked. Default is **false**
password-stacking	optional	Set to **useFirstPass** to indicate that this login module should look for information stored in the LoginContext to use as the identity. This option can be used when stacking other login modules with this one. Default is **false**

Database

This login module is designed to be used for authenticating users against a database backend.

Options	Usage	Description
dsJndiName	required	JNDI name of the datasource containing the tables for users and roles
principalsQuery	required	SQL prepared statement to be executed in order to match the password. Default is **select Password from Principals where PrincipalID=?**
rolesQuery	optional	SQL prepared statement to be executed in order to map roles. It should be an equivalent to **select Role, RoleGroup from Roles where PrincipalID=?**, where Role is the role name and RoleGroup column value should always be "Roles" with capital R.
suspendResume	optional	A boolean flag that specifies that any existing JTA transaction be suspended during DB operations. The default is **true**

Certificate

This login module is designed to authenticate users based on X509Certificates. A use case for this is CLIENT-CERT authentication of a web application.

Options	Usage	Description
securityDomain	optional	Name of the security domain that has the jsse configuration for the truststore holding the trusted certificates
verifier	optional	The class name of the org.jboss.security.auth.certs.X509CertificateVerifier to use for verification of the login certificate

If there is no verifier set, this login module will try to validate the user's certificate with a public certificate stored in the truststore. The public certificate must be stored in the truststore using the DN of the certificate as the truststore alias.

CertificateRoles

This login module extends the Certificate login module to add role mapping capabilities from a properties file. It has the same options plus these additional options:

Options	Usage	Description
rolesProperties	optional	Name of the resource/file containing the roles to assign to each user. Default is **roles.properties**
defaultRolesProperties	optional	Name of the resource/file to fall back to if the rolesProperties file can't be found. Default is **defaultRoles.properties**
roleGroupSeperator	optional	Character to use as the role group separator in the role properties file. Default character is '.' (period)

The role properties file must be in the format username=role1,role2 where the username is the DN of the certificate, escaping any equals and space characters. Here is an example:

```
CN\=unit-tests-client,\ OU\=JBoss\ Inc.,\ O\=JBoss\ Inc.,\ ST\=Washington,\ C\=US=JBossAdmin
```

This would assign the *JBossAdmin* role to an user that presents a certificate with *CN=unit-tests-client, OU=JBoss Inc., O=JBoss Inc., ST=Washington, C=US* as the DN.

DatabaseCertificate

This login module extends the Certificate login to add role mapping capabilities from a database table. It has the same options plus these additional options:

Options	Usage	Description	
dsJndiName	required	JNDI name of the datasource containing the tables for users and roles	
rolesQuery	optional	SQL prepared statement to be executed in order to map roles. It should be an equivalent to **select Role, RoleGroup from Roles where PrincipalID=?**, where Role is the role name and RoleGroup column value should always be "Roles" with capital R. Default is **select Role, RoleGroup from Roles where PrincipalID=?**	
suspendResume	optional	A boolean flag that specifies that any existing JTA transaction be suspended during DB operations. The default is **true**	select Role, RoleGroup from Roles where PrincipalID=?

5.22.28 Simple configuration subsystems

The following subsystems currently have no configuration beyond its root element in the configuration

```
<subsystem xmlns="urn:jboss:domain:jaxrs:1.0"/>
<subsystem xmlns="urn:jboss:domain:jdr:1.0"/>
<subsystem xmlns="urn:jboss:domain:pojo:1.0"/>
<subsystem xmlns="urn:jboss:domain:sar:1.0"/>
```

The presence of each of these turns on a piece of functionality:

Name	Description
`jaxrs`	Enables the deployment and functionality of JAX-RS applications
`jdr`	Enables the gathering of diagnostic data for use in remote analysis of error conditions. Although the data is in a simple format and could be useful to anyone, primarily useful for JBoss EAP subscribers who would provide the data to Red Hat when requesting support
`pojo`	Enables the deployment of applications containing JBoss Microcontainer services, as supported by previous versions of JBoss Application Server
`sar`	Enables the deployment of .SAR archives containing MBean services, as supported by previous versions of JBoss Application Server

5.22.29 Undertow subsystem configuration

> **Web subsystem was replaced in WildFly 8 with Undertow.**

There are two main parts to the undertow subsystem, which are server and Servlet container configuration, as well as some ancillary items. Advanced topics like load balancing and failover are covered on the "High Availability Guide". The default configuration does is suitable for most use cases and provides reasonable performance settings.

Required extension:

```
<extension module="org.wildfly.extension.undertow" />
```

Basic subsystem configuration example:

```
<subsystem xmlns="urn:jboss:domain:undertow:1.0">
        <buffer-caches>
            <buffer-cache name="default" buffer-size="1024" buffers-per-region="1024"
max-regions="10"/>
        </buffer-caches>
        <server name="default-server">
            <http-listener name="default" socket-binding="http" />
            <host name="default-host" alias="localhost">
                <location name="/" handler="welcome-content" />
            </host>
        </server>
        <servlet-container name="default" default-buffer-cache="default"
stack-trace-on-error="local-only" >
            <jsp-config/>
            <persistent-sessions/>
        </servlet-container>
        <handlers>
            <file name="welcome-content" path="${jboss.home.dir}/welcome-content"
directory-listing="true"/>
        </handlers>
    </subsystem>
```

Dependencies on other subsystems:

IO Subsystem

Buffer cache configuration

The buffer cache is used for caching content, such as static files. Multiple buffer caches can be configured, which allows for separate servers to use different sized caches.

Buffers are allocated in regions, and are of a fixed size. If you are caching many small files then using a smaller buffer size will be better.

The total amount of space used can be calculated by multiplying the buffer size by the number of buffers per region by the maximum number of regions.

```
<buffer-caches>
    <buffer-cache name="default" buffer-size="1024" buffers-per-region="1024" max-regions="10"/>
  </buffer-caches>
```

Attribute	Description
buffer-size	The size of the buffers. Smaller buffers allow space to be utilised more effectively
buffers-per-region	The numbers of buffers per region
max-regions	The maximum number of regions. This controls the maximum amount of memory that can be used for caching

Server configuration

A server represents an instance of Undertow. Basically this consists of a set of connectors and some configured handlers.

```
<server name="default-server" default-host="default-host" servlet-container="default" >
```

Attribute	Description
default-host	the virtual host that will be used if an incoming request as no Host: header
servlet-container	the servlet container that will be used by this server, unless is is explicitly overriden by the deployment

Connector configuration

Undertow provides HTTP, HTTPS and AJP connectors, which are configured per server.

Common settings

The following settings are common to all connectors:

Attribute	Description
socket-binding	The socket binding to use. This determines the address and port the listener listens on.
worker	A reference to an XNIO worker, as defined in the IO subsystem. The worker that is in use controls the IO and blocking thread pool.
buffer-pool	A reference to a buffer pool as defined in the IO subsystem. These buffers are used internally to read and write requests. In general these should be at least 8k, unless you are in a memory constrained environment.
enabled	If the connector is enabled.
max-post-size	The maximum size of incoming post requests that is allowed.
buffer-pipelined-data	If responses to HTTP pipelined requests should be buffered, and send out in a single write. This can improve performance if HTTP pipe lining is in use and responses are small.
max-header-size	The maximum size of a HTTP header block that is allowed. Responses with to much data in their header block will have the request terminated and a bad request response send.
max-parameters	The maximum number of query or path parameters that are allowed. This limit exists to prevent hash collision based DOS attacks.
max-headers	The maximum number of headers that are allowed. This limit exists to prevent hash collision based DOS attacks.
max-cookies	The maximum number of cookies that are allowed. This limit exists to prevent hash collision based DOS attacks.
allow-encoded-slash	Set this to true if you want the server to decode percent encoded slash characters. This is probably a bad idea, as it can have security implications, due to different servers interpreting the slash differently. Only enable this if you have a legacy application that requires it.
decode-url	If the URL should be decoded. If this is not set to true then percent encoded characters in the URL will be left as is.
url-charset	The charset to decode the URL to.
always-set-keep-alive	If the 'Connection: keep-alive' header should be added to all responses, even if not required by spec.
disallowed-methods	A comma separated list of HTTP methods that are not allowed. HTTP TRACE is disabled by default.

HTTP Connector

```
<http-listener name="default" socket-binding="http"  />
```

Attribute	Description
certificate-forwarding	If this is set to true then the HTTP listener will read a client certificate from the SSL_CLIENT_CERT header. This allows client cert authentication to be used, even if the server does not have a direct SSL connection to the end user. This should only be enabled for servers behind a proxy that has been configured to always set these headers.
redirect-socket	The socket binding to redirect requests that require security too.
proxy-address-forwarding	If this is enabled then the X-Forwarded-For and X-Forwarded-Proto headers will be used to determine the peer address. This allows applications that are behind a proxy to see the real address of the client, rather than the address of the proxy.

HTTPS listener

Https listener provides secure access to the server. The most important configuration option is security realm which defines SSL secure context.

```
<https-listener name="default" socket-binding="https" security-realm="ssl-realm" />
```

Attribute	Description
security-realm	The security realm to use for the SSL configuration. See Security realm examples for how to configure it: Examples
verify-client	One of either NOT_REQUESTED, REQUESTED or REQUIRED. If client cert auth is in use this should be either REQUESTED or REQUIRED.
enabled-cipher-suites	A list of cypher suit names that are allowed.

AJP listener

```
<ajp-listener name="default" socket-binding="ajp" />
```

Host configuration

The host element corresponds to a virtual host.

Attribute	Description
name	The virtual host name
alias	A whitespace separated list of additional host names that should be matched
default-web-module	The name of a deployment that should be used to serve up requests that do not match anything.

Servlet container configuration

The servlet-container element corresponds to an instance of an Undertow Servlet container. Most servers will only need a single servlet container, however there may be cases where it makes sense to define multiple containers (in particular if you want applications to be isolated, so they cannot dispatch to each other using the RequestDispatcher. You can also use multiple Servlet containers to serve different applications from the same context path on different virtual hosts).

Attribute	Description
allow-non-standard-wrappers	The Servlet specification requires applications to only wrap the request/response using wrapper classes that extend from the ServletRequestWrapper and ServletResponseWrapper classes. If this is set to true then this restriction is relaxed.
default-buffer-cache	The buffer cache that is used to cache static resources in the default Servlet.
stack-trace-on-error	Can be either all, none, or local-only. When set to none Undertow will never display stack traces. When set to All Undertow will always display them (not recommended for production use). When set to local-only Undertow will only display them for requests from local addresses, where there are no headers to indicate that the request has been proxied. Note that this feature means that the Undertow error page will be displayed instead of the default error page specified in web.xml.
default-encoding	The default encoding to use for requests and responses.
use-listener-encoding	If this is true then the default encoding will be the same as that used by the listener that received the request.

JSP configuration

Session Cookie Configuration

This allows you to change the attributes of the session cookie.

Attribute	Description
name	The cookie name
domain	The cookie domain
comment	The cookie comment
http-only	If the cookie is HTTP only
secure	If the cookie is marked secure
max-age	The max age of the cookie

Persistent Session Configuration

Persistent sessions allow session data to be saved across redeploys and restarts. This feature is enabled by adding the persistent-sessions element to the server config. This is mostly intended to be a development time feature.

If the path is not specified then session data is stored in memory, and will only be persistent across redeploys, rather than restarts.

Attribute	Description
path	The path to the persistent sessions data
relative-to	The location that the path is relevant to

AJP listeners

The AJP listeners are child resources of the subsystem undertow. They are used with mod_jk, mod_proxy and mod_cluster of the Apache httpd front-end. Each listener does reference a particular socket binding:

```
[standalone@localhost:9999 /]
/subsystem=undertow/server=default-server:read-children-names(child-type=ajp-listener)
{
    "outcome" => "success",
    "result" => [
        "ajp-listener",
    ]
}

[standalone@localhost:9999 /]
/subsystem=undertow/server=default-server/ajp-listener=*:read-resource(recursive=true)
{
    "outcome" => "success",
    "result" => {
        "enabled" => "true",
        "scheme" => "http",
        "socket-binding" => "ajp",
    }
}
```

Creating a new `ajp-listener` requires you to declare a new `socket binding` first:

```
[standalone@localhost:9999 /]
/socket-binding-group=standard-sockets/socket-binding=ajp:add(port=8009)
```

The newly created, unused socket binding can then be used to create a new connector configuration:

```
[standalone@localhost:9999 /]
/subsystem=undertow/server=default-server/ajp-listener=myListener:add(socket-binding=ajp,
scheme=http, enabled=true)
```

Using Wildfly as a Load Balancer

Wildfly 10 adds support for using the Undertow subsystem as a load balancer. Wildfly supports two different approaches, you can either define a static load balancer, and specify the back end hosts in your configuration, or use it as a mod_cluster frontend, and use mod_cluster to dynamically update the hosts.

General Overview

Wildly uses Undertow's proxy capabilities to act as a load balancer. Undertow will connect to the back end servers using its built in client, and proxies requests.

The following protocols are supported:

- http
- ajp
- http2
- h2c (clear text HTTP2)
  Of these protocols h2c should give the best performance, if the back end servers support it.

The Undertow proxy uses async IO, the only threads that are involved in the request is the IO thread that is responsible for the connection. The connection to the back end server is made from the same thread, which removes the need for any thread safety constructs.
If both the front and back end servers support server push, and HTTP2 is in use then the proxy also supports pushing responses to the client. In cases where proxy and backend are capable of server push, but the client does not support it the server will send a X-Disable-Push header to let the backend know that it should not attempt to push for this request.

Load balancer server profiles

WildFly 11 adds load balancer profiles for both standalone and domain modes.

Example: Start standalone load balancer

```
# configure correct path to WildFly installation
WILDFLY_HOME=/path/to/wildfly
# configure correct IP of the node
MY_IP=192.168.1.1

# run the load balancer profile
$WILDFLY_HOME/bin/standalone.sh -b $MY_IP -bprivate $MY_IP -c standalone-load-balancer.xml
```

It's highly recommended to use private/internal network for communication between load balancer and nodes. To do this set the correct IP address to the `private` interface (`-bprivate` argument).
Example: Start worker node
Run the server with the HA (or Full HA) profile, which has `mod_cluster` component included. If the UDP multicast is working in your environment, the workers should work out of the box without any change. If it's not the case, then configure the IP address of the load-balancer statically.

```
# configure correct path to WildFly installation
WILDFLY_HOME=/path/to/wildfly
# configure correct IP of the node
MY_IP=192.168.1.2

# Configure static load balancer IP address.
# This is necessary when UDP multicast doesn't work in your environment.
LOAD_BALANCER_IP=192.168.1.1
$WILDFLY_HOME/bin/jboss-cli.sh <<EOT
embed-server -c=standalone-ha.xml
/subsystem=modcluster/mod-cluster-config=configuration:write-attribute(name=advertise,value=false)
start the woker node with HA profile
$WILDFLY_HOME/bin/standalone.sh -c standalone-ha.xml -b $MY_IP -bprivate $MY_IP
```

Again, to make it safe, users should configure private/internal IP address into `MY_IP` variable.

Using Wildly as a static load balancer

To use WildFly as a static load balancer the first step is to create a proxy handler in the Undertow subsystem. For the purposes of this example we are going to assume that our load balancer is going to load balance between two servers, sv1.foo.com and sv2.foo.com, and will be using the AJP protocol.

The first step is to add a reverse proxy handler to the Undertow subsystem:

```
/subsystem=undertow/configuration=handler/reverse-proxy=my-handler:add()
```

Then we need to define outbound socket bindings for remote hosts

```
/socket-binding-group=standard-sockets/remote-destination-outbound-socket-binding=remote-host1/:ad
port=8009)
/socket-binding-group=standard-sockets/remote-destination-outbound-socket-binding=remote-host2/:ad
port=8009)
```

and than we add them as hosts to reverse proxy handler

```
/subsystem=undertow/configuration=handler/reverse-proxy=my-handler/host=host1:add(outbound-socket-
scheme=ajp, instance-id=myroute, path=/test)
/subsystem=undertow/configuration=handler/reverse-proxy=my-handler/host=host2:add(outbound-socket-
scheme=ajp, instance-id=myroute, path=/test)
```

Now we need to actually add the reverse proxy to a location. I am going to assume we are serving the path `/app`:

```
/subsystem=undertow/server=default-server/host=default-host/location=\/app:add(handler=my-handler)
```

This is all there is to it. If you point your browser to `http://localhost:8080/app` you should be able to see the proxied content.

The full details of all configuration options available can be found in the subsystem reference.

5.22.30 Web services configuration

JBossWS components are provided to the application server through the webservices subsystem. JBossWS components handle the processing of WS endpoints. The subsystem supports the configuration of published endpoint addresses, and endpoint handler chains. A default webservice subsystem is provided in the server's domain and standalone configuration files.

Structure of the webservices subsystem

Published endpoint address

JBossWS supports the rewriting of the `<soap:address>` element of endpoints published in WSDL contracts. This feature is useful for controlling the server address that is advertised to clients for each endpoint.

The following elements are available and can be modified (all are optional):

Name	Type	Description
modify-wsdl-address	boolean	This boolean enables and disables the address rewrite functionality. When modify-wsdl-address is set to true and the content of <soap:address> is a valid URL, JBossWS will rewrite the URL using the values of wsdl-host and wsdl-port or wsdl-secure-port. When modify-wsdl-address is set to false and the content of <soap:address> is a valid URL, JBossWS will not rewrite the URL. The <soap:address> URL will be used. When the content of <soap:address> is not a valid URL, JBossWS will rewrite it no matter what the setting of modify-wsdl-address. If modify-wsdl-address is set to true and wsdl-host is not defined or explicitly set to '`jbossws.undefined.host`' the content of <soap:address> URL is use. JBossWS uses the requester's host when rewriting the <soap:address> When modify-wsdl-address is not defined JBossWS uses a default value of true.
wsdl-host	string	The hostname / IP address to be used for rewriting `<soap:address>`. If `wsdl-host` is set to `jbossws.undefined.host`, JBossWS uses the requester's host when rewriting the `<soap:address>` When wsdl-host is not defined JBossWS uses a default value of '`jbossws.undefined.host`'.
wsdl-port	int	Set this property to explicitly define the HTTP port that will be used for rewriting the SOAP address. Otherwise the HTTP port will be identified by querying the list of installed HTTP connectors.

wsdl-secure-port	int	Set this property to explicitly define the HTTPS port that will be used for rewriting the SOAP address. Otherwise the HTTPS port will be identified by querying the list of installed HTTPS connectors.
wsdl-uri-scheme	string	This property explicitly sets the URI scheme to use for rewriting `<soap:address>` . Valid values are `http` and `https`. This configuration overrides scheme computed by processing the endpoint (even if a transport guarantee is specified). The provided values for `wsdl-port` and `wsdl-secure-port` (or their default values) are used depending on specified scheme.
wsdl-path-rewrite-rule	string	This string defines a SED substitution command (e.g., 's/regexp/replacement/g') that JBossWS executes against the path component of each <soap:address> URL published from the server. When wsdl-path-rewrite-rule is not defined, JBossWS retains the original path component of each <soap:address> URL. When 'modify-wsdl-address' is set to "false" this element is ignored.

Predefined endpoint configurations

JBossWS enables extra setup configuration data to be predefined and associated with an endpoint implementation. Predefined endpoint configurations can be used for JAX-WS client and JAX-WS endpoint setup. Endpoint configurations can include JAX-WS handlers and key/value properties declarations. This feature provides a convenient way to add handlers to WS endpoints and to set key/value properties that control JBossWS and Apache CXF internals (see Apache CXF configuration).

The webservices subsystem provides schema to support the definition of named sets of endpoint configuration data. Annotation, *org.jboss.ws.api.annotation.EndpointConfig* is provided to map the named configuration to the endpoint implementation.

There is no limit to the number of endpoint configurations that can be defined within the webservices subsystem. Each endpoint configuration must have a name that is unique within the webservices subsystem. Endpoint configurations defined in the webservices subsystem are available for reference by name through the annotation to any endpoint in a deployed application.

WildFly ships with two predefined endpoint configurations. Standard-Endpoint-Config is the default configuration. Recording-Endpoint-Config is an example of custom endpoint configuration and includes a recording handler.

```
[standalone@localhost:9999 /] /subsystem=webservices:read-resource
{
    "outcome" => "success",
    "result" => {
        "endpoint" => {},
        "modify-wsdl-address" => true,
        "wsdl-host" => expression "${jboss.bind.address:127.0.0.1}",
        "endpoint-config" => {
            "Standard-Endpoint-Config" => undefined,
            "Recording-Endpoint-Config" => undefined
        }
    }
}
```

The `Standard-Endpoint-Config` is a special endpoint configuration. It is used for any endpoint that does not have an explicitly assigned endpoint configuration.

Endpoint configs

Endpoint configs are defined using `the endpoint-config` element. Each endpoint configuration may include properties and handlers set to the endpoints associated to the configuration.

```
[standalone@localhost:9999 /]
/subsystem=webservices/endpoint-config=Recording-Endpoint-Config:read-resource
{
    "outcome" => "success",
    "result" => {
        "post-handler-chain" => undefined,
        "property" => undefined,
        "pre-handler-chain" => {"recording-handlers" => undefined}
    }
}
```

A new endpoint configuration can be added as follows:

```
[standalone@localhost:9999 /] /subsystem=webservices/endpoint-config=My-Endpoint-Config:add
{
    "outcome" => "success",
    "response-headers" => {
        "operation-requires-restart" => true,
        "process-state" => "restart-required"
    }
}
```

Handler chains

Each endpoint configuration may be associated with zero or more PRE and POST handler chains. Each handler chain may include JAXWS handlers. For outbound messages the PRE handler chains are executed before any handler that is attached to the endpoint using the standard means, such as with annotation @HandlerChain, and POST handler chains are executed after those objects have executed. For inbound messages the POST handler chains are executed before any handler that is attached to the endpoint using the standard means and the PRE handler chains are executed after those objects have executed.

```
* Server inbound messages
Client --> ... --> POST HANDLER --> ENDPOINT HANDLERS --> PRE HANDLERS --> Endpoint

* Server outbound messages
Endpoint --> PRE HANDLER --> ENDPOINT HANDLERS --> POST HANDLERS --> ... --> Client
```

The protocol-binding attribute must be used to set the protocols for which the chain will be triggered.

```
[standalone@localhost:9999 /]
/subsystem=webservices/endpoint-config=Recording-Endpoint-Config/pre-handler-chain=recording-handl
"outcome" => "success",
    "result" => {
        "protocol-bindings" => "##SOAP11_HTTP ##SOAP11_HTTP_MTOM ##SOAP12_HTTP
##SOAP12_HTTP_MTOM",
        "handler" => {"RecordingHandler" => undefined}
    },
    "response-headers" => {"process-state" => "restart-required"}
}
```

A new handler chain can be added as follows:

```
[standalone@localhost:9999 /]
/subsystem=webservices/endpoint-config=My-Endpoint-Config/post-handler-chain=my-handlers:add(proto
"outcome" => "success",
    "response-headers" => {
        "operation-requires-restart" => true,
        "process-state" => "restart-required"
    }
}
[standalone@localhost:9999 /]
/subsystem=webservices/endpoint-config=My-Endpoint-Config/post-handler-chain=my-handlers:read-reso
"outcome" => "success",
    "result" => {
        "handler" => undefined,
        "protocol-bindings" => "##SOAP11_HTTP"
    },
    "response-headers" => {"process-state" => "restart-required"}
}
```

Handlers

JAXWS handler can be added in handler chains:

```
[standalone@localhost:9999 /]
/subsystem=webservices/endpoint-config=Recording-Endpoint-Config/pre-handler-chain=recording-handl
"outcome" => "success",
    "result" => {"class" => "org.jboss.ws.common.invocation.RecordingServerHandler"},
    "response-headers" => {"process-state" => "restart-required"}
}
[standalone@localhost:9999 /]
/subsystem=webservices/endpoint-config=My-Endpoint-Config/post-handler-chain=my-handlers/handler=f
"outcome" => "success",
    "response-headers" => {
        "operation-requires-restart" => true,
        "process-state" => "restart-required"
    }
}
```

Endpoint-config handler classloading

The `class` attribute is used to provide the fully qualified class name of the handler. At deploy time, an instance of the class is created for each referencing deployment. For class creation to succeed, the deployment classloader must to be able to load the handler class.

Runtime information

Each web service endpoint is exposed through the deployment that provides the endpoint implementation. Each endpoint can be queried as a deployment resource. For further information please consult the chapter "Application Deployment". Each web service endpoint specifies a web context and a WSDL Url:

```
[standalone@localhost:9999 /] /deployment="*"/subsystem=webservices/endpoint="*":read-resource
{
   "outcome" => "success",
   "result" => [{
       "address" => [
           ("deployment" => "jaxws-samples-handlerchain.war"),
           ("subsystem" => "webservices"),
           ("endpoint" => "jaxws-samples-handlerchain:TestService")
       ],
       "outcome" => "success",
       "result" => {
           "class" => "org.jboss.test.ws.jaxws.samples.handlerchain.EndpointImpl",
           "context" => "jaxws-samples-handlerchain",
           "name" => "TestService",
           "type" => "JAXWS_JSE",
           "wsdl-url" => "http://localhost:8080/jaxws-samples-handlerchain?wsdl"
       }
   }]
}
```

Component Reference

The web service subsystem is provided by the JBossWS project. For a detailed description of the available configuration properties, please consult the project documentation.

- JBossWS homepage: http://www.jboss.org/jbossws
- Project Documentation: https://docs.jboss.org/author/display/JBWS

5.23 Target Audience

This document is a guide to the setup, administration, and configuration of WildFly.

5.23.1 Prerequisites

Before continuing, you should know how to download, install and run WildFly. For more information on these steps, refer here: Getting Started Guide.

5.23.2 Examples in this guide

The examples in this guide are largely expressed as XML configuration file excerpts, or by using a representation of the de-typed management model.

6 Developer Guide

6.1 WildFly Developer Guide

6.1.1 Target Audience

Java Developers

6.1.2 Prerequisites

6.2 Class loading in WildFly

Since JBoss AS 7, Class loading is considerably different to previous versions of JBoss AS. Class loading is based on the JBoss Modules project. Instead of the more familiar hierarchical class loading environment, WildFly's class loading is based on modules that have to define explicit dependencies on other modules. Deployments in WildFly are also modules, and do not have access to classes that are defined in jars in the application server unless an explicit dependency on those classes is defined.

6.2.1 Deployment Module Names

Module names for top level deployments follow the format `deployment.myarchive.war` while sub deployments are named like `deployment.myear.ear.mywar.war`.

This means that it is possible for a deployment to import classes from another deployment using the other deployments module name, the details of how to add an explicit module dependency are explained below.

6.2.2 Automatic Dependencies

Even though in WildFly modules are isolated by default, as part of the deployment process some dependencies on modules defined by the application server are set up for you automatically. For instance, if you are deploying a Java EE application a dependency on the Java EE API's will be added to your module automatically. Similarly if your module contains a beans.xml file a dependency on Weld will be added automatically, along with any supporting modules that weld needs to operate.

For a complete list of the automatic dependencies that are added, please see Implicit module dependencies for deployments.

Automatic dependencies can be excluded through the use of `jboss-deployment-structure.xml`.

6.2.3 Class Loading Precedence

A common source of errors in Java applications is including API classes in a deployment that are also provided by the container. This can result in multiple versions of the class being created and the deployment failing to deploy properly. To prevent this in WildFly, module dependencies are added in a specific order that should prevent this situation from occurring.

In order of highest priority to lowest priority

1. System Dependencies - These are dependencies that are added to the module automatically by the container, including the Java EE api's.
2. User Dependencies - These are dependencies that are added through `jboss-deployment-structure.xml` or through the `Dependencies:` manifest entry.
3. Local Resource - Class files packaged up inside the deployment itself, e.g. class files from `WEB-INF/classes` or `WEB-INF/lib` of a war.
4. Inter deployment dependencies - These are dependencies on other deployments in an ear deployment. This can include classes in an ear's lib directory, or classes defined in other ejb jars.

6.2.4 WAR Class Loading

The war is considered to be a single module, so classes defined in `WEB-INF/lib` are treated the same as classes in `WEB-INF/classes`. All classes packaged in the war will be loaded with the same class loader.

6.2.5 EAR Class Loading

Ear deployments are multi-module deployments. This means that not all classes inside an ear will necessarily have access to all other classes in the ear, unless explicit dependencies have been defined. By default the `EAR/lib` directory is a single module, and every WAR or EJB jar deployment is also a separate module. Sub deployments (wars and ejb-jars) always have a dependency on the parent module, which gives them access to classes in `EAR/lib`, however they do not always have an automatic dependency on each other. This behaviour is controlled via the `ear-subdeployments-isolated` setting in the ee subsystem configuration:

```
<subsystem xmlns="urn:jboss:domain:ee:1.0" >
  <ear-subdeployments-isolated>false</ear-subdeployments-isolated>
</subsystem>
```

By default this is set to false, which allows the sub-deployments to see classes belonging to other sub-deployments within the .ear.

For example, consider the following .ear deployment:

```
myapp.ear
 |
 |--- web.war
 |
 |--- ejb1.jar
 |
 |--- ejb2.jar
```

If the ear-subdeployments-isolated is set to false, then the classes in web.war can access classes belonging to ejb1.jar and ejb2.jar. Similarly, classes from ejb1.jar can access classes from ejb2.jar (and vice-versa).

The ear-subdeployments-isolated element value has no effect on the isolated classloader of the .war file(s). i.e. irrespective of whether this flag is set to true or false, the .war within a .ear will have a isolated classloader and other sub-deployments within that .ear will not be able to access classes from that .war. This is as per spec.

If the ear-subdeployments-isolated is set to true then no automatic module dependencies between the sub-deployments are set up. User must manually setup the dependency with `Class-Path` entries, or by setting up explicit module dependencies.

Portability

The Java EE specification says that portable applications should not rely on sub deployments having access to other sub deployments unless an explicit Class-Path entry is set in the MANIFEST.MF. So portable applications should always use Class-Path entry to explicitly state their dependencies.

It is also possible to override the ear-subdeployments-isolated element value at a per deployment level. See the section on jboss-deployment-structure.xml below.

`Dependencies:` Manifest Entries

Deployments (or more correctly modules within a deployment) may set up dependencies on other modules by adding a `Dependencies:` manifest entry. This entry consists of a comma separated list of module names that the deployment requires. The available modules can be seen under the `modules` directory in the application server distribution. For example to add a dependency on javassist and apache velocity you can add a manifest entry as follows:

```
Dependencies: org.javassist export,org.apache.velocity export services,org.antlr
```

Each dependency entry may also specify some of the following parameters by adding them after the module name:

- `export` This means that the dependencies will be exported, so any module that depends on this module will also get access to the dependency.
- `services` By default items in META-INF of a dependency are not accessible, this makes items from `META-INF/services` accessible so services in the modules can be loaded.
- `optional` If this is specified the deployment will not fail if the module is not available.
- `meta-inf` This will make the contents of the `META-INF` directory available (unlike `services`, which just makes `META-INF/services` available). In general this will not cause any deployment descriptors in META-INF to be processed, with the exception of `beans.xml`. If a `beans.xml` file is present this module will be scanned by Weld and any resulting beans will be available to the application.
- `annotations` If a jandex index has be created for the module these annotations will be merged into the deployments annotation index. The Jandex index can be generated using the Jandex ant task , and must be named `META-INF/jandex.idx`. Note that it is not necessary to break open the jar being indexed to add this to the modules class path, a better approach is to create a jar containing just this index, and adding it as an additional resource root in the `module.xml` file.

Adding a dependency to all modules in an EAR

Using the `export` parameter it is possible to add a dependency to all sub deployments in an ear. If a module is exported from a `Dependencies:` entry in the top level of the ear (or by a jar in the `ear/lib` directory) it will be available to all sub deployments as well.

To generate a MANIFEST.MF entry when using maven put the following in your pom.xml:

pom.xml

```
<build>
   ...
   <plugins>
     <plugin>
       <groupId>org.apache.maven.plugins</groupId>
       <artifactId>maven-war-plugin</artifactId>
       <configuration>
          <archive>
             <manifestEntries>
                <Dependencies>org.slf4j</Dependencies>
             </manifestEntries>
          </archive>
       </configuration>
     </plugin>
   </plugins>
</build>
```

If your deployment is a jar you must use the `maven-jar-plugin` rather than the `maven-war-plugin`.

Class Path Entries

It is also possible to add module dependencies on other modules inside the deployment using the `Class-Path` manifest entry. This can be used within an ear to set up dependencies between sub deployments, and also to allow modules access to additional jars deployed in an ear that are not sub deployments and are not in the `EAR/lib` directory. If a jar in the `EAR/lib` directory references a jar via `Class-Path:` then this additional jar is merged into the parent ear's module, and is accessible to all sub deployments in the ear.

6.2.6 Global Modules

It is also possible to set up global modules, that are accessible to all deployments. This is done by modifying the configuration file (standalone/domain.xml).

For example, to add javassist to all deployments you can use the following XML:

standalone.xml/domain.xml

```
<subsystem xmlns="urn:jboss:domain:ee:1.0" >
  <global-modules>
    <module name="org.javassist" slot="main" />
  </global-modules>
</subsystem>
```

Note that the `slot` field is optional and defaults to `main`.

6.2.7 JBoss Deployment Structure File

`jboss-deployment-structure.xml` is a JBoss specific deployment descriptor that can be used to control class loading in a fine grained manner. It should be placed in the top level deployment, in `META-INF` (or `WEB-INF` for web deployments). It can do the following:

- Prevent automatic dependencies from being added
- Add additional dependencies
- Define additional modules
- Change an EAR deployments isolated class loading behaviour
- Add additional resource roots to a module

An example of a complete `jboss-deployment-structure.xml` file for an ear deployment is as follows:

jboss-deployment-structure.xml

```
<jboss-deployment-structure>
  <!-- Make sub deployments isolated by default, so they cannot see each others classes without
a Class-Path entry -->
  <ear-subdeployments-isolated>true</ear-subdeployments-isolated>
  <!-- This corresponds to the top level deployment. For a war this is the war's module, for an
ear -->
  <!-- This is the top level ear module, which contains all the classes in the EAR's lib folder
-->
  <deployment>
     <!-- exclude-subsystem prevents a subsystems deployment unit processors running on a
deployment -->
     <!-- which gives basically the same effect as removing the subsystem, but it only affects
single deployment -->
     <exclude-subsystems>
        <subsystem name="resteasy" />
```

```
    </exclude-subsystems>
    <!-- Exclusions allow you to prevent the server from automatically adding some dependencies
-->
    <exclusions>
        <module name="org.javassist" />
    </exclusions>
    <!-- This allows you to define additional dependencies, it is the same as using the
Dependencies: manifest attribute -->
    <dependencies>
      <module name="deployment.javassist.proxy" />
      <module name="deployment.myjavassist" />
      <!-- Import META-INF/services for ServiceLoader impls as well -->
      <module name="myservicemodule" services="import"/>
    </dependencies>
    <!-- These add additional classes to the module. In this case it is the same as including
the jar in the EAR's lib directory -->
    <resources>
      <resource-root path="my-library.jar" />
    </resources>
  </deployment>
  <sub-deployment name="myapp.war">
    <!-- This corresponds to the module for a web deployment -->
    <!-- it can use all the same tags as the <deployment> entry above -->
    <dependencies>
      <!-- Adds a dependency on a ejb jar. This could also be done with a Class-Path entry -->
      <module name="deployment.myear.ear.myejbjar.jar" />
    </dependencies>
    <!-- Set's local resources to have the lowest priority -->
    <!-- If the same class is both in the sub deployment and in another sub deployment that -->
    <!-- is visible to the war, then the Class from the other deployment will be loaded,  -->
    <!-- rather than the class actually packaged in the war. -->
    <!-- This can be used to resolve ClassCastExceptions  if the same class is in multiple sub
deployments-->
    <local-last value="true" />
  </sub-deployment>
  <!-- Now we are going to define two additional modules -->
  <!-- This one is a different version of javassist that we have packaged -->
  <module name="deployment.myjavassist" >
    <resources>
     <resource-root path="javassist.jar" >
       <!-- We want to use the servers version of javassist.util.proxy.* so we filter it out-->
       <filter>
         <exclude path="javassist/util/proxy" />
       </filter>
     </resource-root>
    </resources>
  </module>
  <!-- This is a module that re-exports the containers version of javassist.util.proxy -->
  <!-- This means that there is only one version of the Proxy classes defined          -->
  <module name="deployment.javassist.proxy" >
    <dependencies>
      <module name="org.javassist" >
        <imports>
          <include path="javassist/util/proxy" />
          <exclude path="/**" />
        </imports>
      </module>
    </dependencies>
```

```
  </module>
</jboss-deployment-structure>
```

The xsd for jboss-deployment-structure.xml is available at https://github.com/wildfly/wildfly/blob/master/build/src/main/resources/docs/schema/jboss-deployment-

6.2.8 Accessing JDK classes

Not all JDK classes are exposed to a deployment by default. If your deployment uses JDK classes that are not exposed you can get access to them using jboss-deployment-structure.xml with system dependencies:

Using jboss-deployment-structure.xml to access JDK classes

```
<jboss-deployment-structure xmlns="urn:jboss:deployment-structure:1.1">
    <deployment>
        <dependencies>
            <system export="true">
                <paths>
                    <path name="com/sun/corba/se/spi/legacy/connection"/>
                </paths>
            </system>
        </dependencies>
    </deployment>
</jboss-deployment-structure>
```

6.2.9 The "jboss.api" property and application use of modules shipped with WildFly

The WildFly distribution includes a large number of modules, a great many of which are included for use by WildFly internals, with no testing of the appropriateness of their direct use by applications or any commitment to continue to ship those modules in future releases if they are no longer needed by the internals. So how can a user know whether it is advisable for their application to specify an explicit dependency on a module WildFly ships? The "jboss.api" property specified in the module's module.xml file can tell you:

Example declaration of the jboss.api property

```
<module xmlns="urn:jboss:module:1.3" name="com.google.guava">
    <properties>
        <property name="jboss.api" value="private"/>
    </properties>
```

If a module does not have a property element like the above, then it's equivalent to one with a value of "public".

Following are the meanings of the various values you may see for the jboss.api property:

Value	Meaning
public	May be explicitly depended upon by end user applications. Will continue to be available in future releases within the same major series and should not have incompatible API changes in future releases within the same minor series, and ideally not within the same major series.
private	Intended for internal use only. Only tested according to internal usage. May not be safe for end user applications to use directly. Could change significantly or be removed in a future release without notice.
unsupported	If you see this value in a module.xml in a WildFly release, please file a bug report, as it is not applicable in WildFly. In EAP it has a meaning equivalent to "private" but that does not mean the module is "private" in WildFly; it could very easily be "public".
preview	May be explicitly depended upon by end user applications, but there are no guarantees of continued availability in future releases or that there will not be incompatible API changes. This is not a common classification in WildFly. It is not used in WildFly 10.
deprecated	May be explicitly depended upon by end user applications. Stable and reliable but an alternative should be sought. Will be removed in a future major release.

Note that these definitions are only applicable to WildFly. In EAP and other Red Hat products based on WildFly the same classifiers are used, with generally similar meaning, but the precise meaning is per the definitions on the Red Hat customer support portal.

If an application declares a direct dependency on a module marked "private", "unsupported" or "deprecated", during deployment a WARN message will be logged. The logging will be in log categories "org.jboss.as.dependency.private", "org.jboss.as.dependency.unsupported" and "org.jboss.as.dependency.deprecated" respectively. These categories are not used for other purposes, so once you feel sufficiently warned the logging can be safely suppressed by turning the log level for the relevant category to ERROR or higher.

Other than the WARN messages noted above, declaring a direct dependency on a non-public module has no impact on how WildFly processes the deployment.

6.3 Implicit module dependencies for deployments

As explained in the Class Loading in WildFly article, WildFly 8 is based on module classloading. A class within a module B isn't visible to a class within a module A, unless module B adds a dependency on module A. Module dependencies can be explicitly (as explained in that classloading article) or can be "implicit". This article will explain what implicit module dependencies mean and how, when and which modules are added as implicit dependencies.

6.3.1 What's an implicit module dependency?

Consider an application deployment which contains EJBs. EJBs typically need access to classes from the javax.ejb.* package and other Java EE API packages. The jars containing these packages are already shipped in WildFly and are available as "modules". The module which contains the javax.ejb.* classes has a specific name and so does the module which contains all the Java EE API classes. For an application to be able to use these classes, it has to add a dependency on the relevant modules. Forcing the application developers to add module dependencies like these (i.e. dependencies which can be "inferred") isn't a productive approach. Hence, whenever an application is being deployed, the deployers within the server, which are processing this deployment "implicitly" add these module dependencies to the deployment so that these classes are visible to the deployment at runtime. This way the application developer doesn't have to worry about adding them explicitly. How and when these implicit dependencies are added is explained in the next section.

6.3.2 How and when is an implicit module dependency added?

When a deployment is being processed by the server, it goes through a chain of "deployment processors". Each of these processors will have a way to check if the deployment meets a certain criteria and if it does, the deployment processor adds a implicit module dependency to that deployment. Let's take an example - Consider (again) an EJB3 deployment which has the following class:

MySuperDuperBean.java

```
@Stateless
public class MySuperDuperBean {

...

}
```

As can be seen, we have a simple @Stateless EJB. When the deployment containing this class is being processed, the EJB deployment processor will see that the deployment contains a class with the @Stateless annotation and thus identifies this as a EJB deployment. **This is just one of the several ways, various deployment processors can identify a deployment of some specific type.** The EJB deployment processor will then add an implicit dependency on the Java EE API module, so that all the Java EE API classes are visible to the deployment.

Some subsystems will always add a API classes, even if the trigger condition is not met. These are listed separately below.

In the next section, we'll list down the implicit module dependencies that are added to a deployment, by various deployers within WildFly.

6.3.3 Which are the implicit module dependencies?

Subsystem responsible for adding the implicit dependency	Dependencies that are always added	Dependencies that are added if a trigger condition is met
Core Server	• javax.api • sun.jdk • org.jboss.vfs	
Batch Subsystem	• javax.batch.api	
EE Subsystem	• javaee.api	
EJB3 subsystem		• javaee.api
JAX-RS (Resteasy) subsystem	• javax.xml.bind.api	• org.jboss.resteasy.resteasy-atom-provider • org.jboss.resteasy.resteasy-cdi • org.jboss.resteasy.resteasy-jaxrs • org.jboss.resteasy.resteasy-jaxb-provider • org.jboss.resteasy.resteasy-jackson-provider • org.jboss.resteasy.resteasy-jsapi • org.jboss.resteasy.resteasy-multipart-provide • org.jboss.resteasy.async-http-servlet-30

JCA subsystem	• javax.resource.api	• javax.jms.api • javax.validation.api • org.jboss.logging • org.jboss.ironjacamar.api • org.jboss.ironjacamar.impl • org.hibernate.validator
JPA (Hibernate) subsystem	• javax.persistence.api	• javaee.api • org.jboss.as.jpa • org.hibernate
Logging Subsystem	• org.jboss.logging • org.apache.commons.logging • org.apache.log4j • org.slf4j • org.jboss.logging.jul-to-slf4j-stub	
SAR Subsystem		• org.jboss.logging • org.jboss.modules
Security Subsystem	• org.picketbox	
Web Subsystem		• javaee.api • com.sun.jsf-impl • org.hibernate.validator • org.jboss.as.web • org.jboss.logging
Web Services Subsystem	• org.jboss.ws.api • org.jboss.ws.spi	

Weld (CDI) Subsystem		• javax.persistence.api • javaee.api • org.javassist • org.jboss.interceptor • org.jboss.as.weld • org.jboss.logging • org.jboss.weld.core • org.jboss.weld.api • org.jboss.weld.spi

6.4 How do I migrate my application from JBoss AS 5 or AS 6 to WildFly?

Couldn't find a page to include called: How do I migrate my application from AS5 or AS6 to WildFly

6.5 EJB invocations from a remote standalone client using JNDI

This chapter explains how to invoke EJBs from a remote client by using the JNDI API to first lookup the bean proxy and then invoke on that proxy.

After you have read this article, do remember to take a look at Remote EJB invocations via JNDI - EJB client API or remote-naming project

Before getting into the details, we would like the users to know that we have introduced a new EJB client API, which is a WildFly-specific API and allows invocation on remote EJBs. This client API isn't based on JNDI. So remote clients need not rely on JNDI API to invoke on EJBs. A separate document covering the EJB remote client API will be made available. For now, you can refer to the javadocs of the EJB client project at http://docs.jboss.org/ejbclient/. In this document, we'll just concentrate on the traditional JNDI based invocation on EJBs. So let's get started:

6.5.1 Deploying your EJBs on the server side:

Users who already have EJBs deployed on the server side can just skip to the next section.

As a first step, you'll have to deploy your application containing the EJBs on the Wildfly server. If you want those EJBs to be remotely invocable, then you'll have to expose at least one remote view for that bean. In this example, let's consider a simple Calculator stateless bean which exposes a RemoteCalculator remote business interface. We'll also have a simple stateful CounterBean which exposes a RemoteCounter remote business interface. Here's the code:

```
package org.jboss.as.quickstarts.ejb.remote.stateless;

/**
 * @author Jaikiran Pai
 */
public interface RemoteCalculator {

    int add(int a, int b);

    int subtract(int a, int b);
}
```

```
package org.jboss.as.quickstarts.ejb.remote.stateless;

import javax.ejb.Remote;
import javax.ejb.Stateless;

/**
 * @author Jaikiran Pai
 */
@Stateless
@Remote(RemoteCalculator.class)
public class CalculatorBean implements RemoteCalculator {

    @Override
    public int add(int a, int b) {
        return a + b;
    }

    @Override
    public int subtract(int a, int b) {
        return a - b;
    }
}
```

```
package org.jboss.as.quickstarts.ejb.remote.stateful;

/**
 * @author Jaikiran Pai
 */
public interface RemoteCounter {

    void increment();

    void decrement();

    int getCount();
}
```

```
package org.jboss.as.quickstarts.ejb.remote.stateful;

import javax.ejb.Remote;
import javax.ejb.Stateful;

/**
 * @author Jaikiran Pai
 */
@Stateful
@Remote(RemoteCounter.class)
public class CounterBean implements RemoteCounter {

    private int count = 0;

    @Override
    public void increment() {
        this.count++;
    }

    @Override
    public void decrement() {
        this.count--;
    }

    @Override
    public int getCount() {
        return this.count;
    }
}
```

Let's package this in a jar (how you package it in a jar is out of scope of this chapter) named "jboss-as-ejb-remote-app.jar" and deploy it to the server. Make sure that your deployment has been processed successfully and there aren't any errors.

6.5.2 Writing a remote client application for accessing and invoking the EJBs deployed on the server

The next step is to write an application which will invoke the EJBs that you deployed on the server. In WildFly, you can either choose to use the WildFly specific EJB client API to do the invocation or use JNDI to lookup a proxy for your bean and invoke on that returned proxy. In this chapter we will concentrate on the JNDI lookup and invocation and will leave the EJB client API for a separate chapter.

So let's take a look at what the client code looks like for looking up the JNDI proxy and invoking on it. Here's the entire client code which invokes on a stateless bean:

```
package org.jboss.as.quickstarts.ejb.remote.client;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.security.Security;
import java.util.Hashtable;

import org.jboss.as.quickstarts.ejb.remote.stateful.CounterBean;
import org.jboss.as.quickstarts.ejb.remote.stateful.RemoteCounter;
import org.jboss.as.quickstarts.ejb.remote.stateless.CalculatorBean;
import org.jboss.as.quickstarts.ejb.remote.stateless.RemoteCalculator;
import org.jboss.sasl.JBossSaslProvider;

/**
 * A sample program which acts a remote client for a EJB deployed on Wildfly 10 server.
 * This program shows how to lookup stateful and stateless beans via JNDI and then invoke on
them
 *
 * @author Jaikiran Pai
 */
public class RemoteEJBClient {

    public static void main(String[] args) throws Exception {
        // Invoke a stateless bean
        invokeStatelessBean();

        // Invoke a stateful bean
        invokeStatefulBean();
    }

    /**
     * Looks up a stateless bean and invokes on it
     *
     * @throws NamingException
     */
    private static void invokeStatelessBean() throws NamingException {
        // Let's lookup the remote stateless calculator
        final RemoteCalculator statelessRemoteCalculator = lookupRemoteStatelessCalculator();
        System.out.println("Obtained a remote stateless calculator for invocation");
        // invoke on the remote calculator
```

```
        int a = 204;
        int b = 340;
        System.out.println("Adding " + a + " and " + b + " via the remote stateless calculator
deployed on the server");
        int sum = statelessRemoteCalculator.add(a, b);
        System.out.println("Remote calculator returned sum = " + sum);
        if (sum != a + b) {
            throw new RuntimeException("Remote stateless calculator returned an incorrect sum "
+ sum + " ,expected sum was " + (a + b));
        }
        // try one more invocation, this time for subtraction
        int num1 = 3434;
        int num2 = 2332;
        System.out.println("Subtracting " + num2 + " from " + num1 + " via the remote stateless
calculator deployed on the server");
        int difference = statelessRemoteCalculator.subtract(num1, num2);
        System.out.println("Remote calculator returned difference = " + difference);
        if (difference != num1 - num2) {
            throw new RuntimeException("Remote stateless calculator returned an incorrect
difference " + difference + " ,expected difference was " + (num1 - num2));
        }
    }

    /**
     * Looks up a stateful bean and invokes on it
     *
     * @throws NamingException
     */
    private static void invokeStatefulBean() throws NamingException {
        // Let's lookup the remote stateful counter
        final RemoteCounter statefulRemoteCounter = lookupRemoteStatefulCounter();
        System.out.println("Obtained a remote stateful counter for invocation");
        // invoke on the remote counter bean
        final int NUM_TIMES = 20;
        System.out.println("Counter will now be incremented " + NUM_TIMES + " times");
        for (int i = 0; i < NUM_TIMES; i++) {
            System.out.println("Incrementing counter");
            statefulRemoteCounter.increment();
            System.out.println("Count after increment is " + statefulRemoteCounter.getCount());
        }
        // now decrementing
        System.out.println("Counter will now be decremented " + NUM_TIMES + " times");
        for (int i = NUM_TIMES; i > 0; i--) {
            System.out.println("Decrementing counter");
            statefulRemoteCounter.decrement();
            System.out.println("Count after decrement is " + statefulRemoteCounter.getCount());
        }
    }

    /**
     * Looks up and returns the proxy to remote stateless calculator bean
     *
     * @return
     * @throws NamingException
     */
    private static RemoteCalculator lookupRemoteStatelessCalculator() throws NamingException {
        final Hashtable jndiProperties = new Hashtable();
        jndiProperties.put(Context.URL_PKG_PREFIXES, "org.jboss.ejb.client.naming");
```

```
        final Context context = new InitialContext(jndiProperties);
        // The app name is the application name of the deployed EJBs. This is typically the ear
name
        // without the .ear suffix. However, the application name could be overridden in the
application.xml of the
        // EJB deployment on the server.
        // Since we haven't deployed the application as a .ear, the app name for us will be an
empty string
        final String appName = "";
        // This is the module name of the deployed EJBs on the server. This is typically the jar
name of the
        // EJB deployment, without the .jar suffix, but can be overridden via the ejb-jar.xml
        // In this example, we have deployed the EJBs in a jboss-as-ejb-remote-app.jar, so the
module name is
        // jboss-as-ejb-remote-app
        final String moduleName = "jboss-as-ejb-remote-app";
        // AS7 allows each deployment to have an (optional) distinct name. We haven't specified
a distinct name for
        // our EJB deployment, so this is an empty string
        final String distinctName = "";
        // The EJB name which by default is the simple class name of the bean implementation
class
        final String beanName = CalculatorBean.class.getSimpleName();
        // the remote view fully qualified class name
        final String viewClassName = RemoteCalculator.class.getName();
        // let's do the lookup
        return (RemoteCalculator) context.lookup("ejb:" + appName + "/" + moduleName + "/" +
distinctName + "/" + beanName + "!" + viewClassName);
    }

    /**
     * Looks up and returns the proxy to remote stateful counter bean
     *
     * @return
     * @throws NamingException
     */
    private static RemoteCounter lookupRemoteStatefulCounter() throws NamingException {
        final Hashtable jndiProperties = new Hashtable();
        jndiProperties.put(Context.URL_PKG_PREFIXES, "org.jboss.ejb.client.naming");
        final Context context = new InitialContext(jndiProperties);
        // The app name is the application name of the deployed EJBs. This is typically the ear
name
        // without the .ear suffix. However, the application name could be overridden in the
application.xml of the
        // EJB deployment on the server.
        // Since we haven't deployed the application as a .ear, the app name for us will be an
empty string
        final String appName = "";
        // This is the module name of the deployed EJBs on the server. This is typically the jar
name of the
        // EJB deployment, without the .jar suffix, but can be overridden via the ejb-jar.xml
        // In this example, we have deployed the EJBs in a jboss-as-ejb-remote-app.jar, so the
module name is
        // jboss-as-ejb-remote-app
        final String moduleName = "jboss-as-ejb-remote-app";
        // AS7 allows each deployment to have an (optional) distinct name. We haven't specified
a distinct name for
        // our EJB deployment, so this is an empty string
```

```
        final String distinctName = "";
        // The EJB name which by default is the simple class name of the bean implementation
class
        final String beanName = CounterBean.class.getSimpleName();
        // the remote view fully qualified class name
        final String viewClassName = RemoteCounter.class.getName();
        // let's do the lookup (notice the ?stateful string as the last part of the jndi name
for stateful bean lookup)
        return (RemoteCounter) context.lookup("ejb:" + appName + "/" + moduleName + "/" +
distinctName + "/" + beanName + "!" + viewClassName + "?stateful");
    }
}
```

The entire server side and client side code is hosted at the github repo here ejb-remote

The code has some comments which will help you understand each of those lines. But we'll explain here in more detail what the code does. As a first step in the client code, we'll do a lookup of the EJB using a JNDI name. In AS7, for remote access to EJBs, you use the ejb: namespace with the following syntax:

For stateless beans:

```
ejb:<app-name>/<module-name>/<distinct-name>/<bean-name>!<fully-qualified-classname-of-the-remote-
```

For stateful beans:

```
ejb:<app-name>/<module-name>/<distinct-name>/<bean-name>!<fully-qualified-classname-of-the-remote-
```

The ejb: namespace identifies it as a EJB lookup and is a constant (i.e. doesn't change) for doing EJB lookups. The rest of the parts in the jndi name are as follows:

app-name : This is the name of the .ear (without the .ear suffix) that you have deployed on the server and contains your EJBs.

- Java EE 6 allows you to override the application name, to a name of your choice by setting it in the application.xml. If the deployment uses uses such an override then the app-name used in the JNDI name should match that name.
- EJBs can also be deployed in a .war or a plain .jar (like we did in step 1). In such cases where the deployment isn't an .ear file, then the app-name must be an empty string, while doing the lookup.

module-name : This is the name of the .jar (without the .jar suffix) that you have deployed on the server and the contains your EJBs. If the EJBs are deployed in a .war then the module name is the .war name (without the .war suffix).

- Java EE 6 allows you to override the module name, by setting it in the ejb-jar.xml/web.xml of your deployment. If the deployment uses such an override then the module-name used in the JNDI name should match that name.
- Module name part cannot be an empty string in the JNDI name

distinct-name : This is a WildFly-specific name which can be optionally assigned to the deployments that are deployed on the server. More about the purpose and usage of this will be explained in a separate chapter. If a deployment doesn't use distinct-name then, use an empty string in the JNDI name, for distinct-name

bean-name : This is the name of the bean for which you are doing the lookup. The bean name is typically the unqualified classname of the bean implementation class, but can be overriden through either ejb-jar.xml or via annotations. The bean name part cannot be an empty string in the JNDI name.

fully-qualified-classname-of-the-remote-interface : This is the fully qualified class name of the interface for which you are doing the lookup. The interface should be one of the remote interfaces exposed by the bean on the server. The fully qualified class name part cannot be an empty string in the JNDI name.

For stateful beans, the JNDI name expects an additional "?stateful" to be appended after the fully qualified interface name part. This is because for stateful beans, a new session gets created on JNDI lookup and the EJB client API implementation doesn't contact the server during the JNDI lookup to know what kind of a bean the JNDI name represents (we'll come to this in a while). So the JNDI name itself is expected to indicate that the client is looking up a stateful bean, so that an appropriate session can be created.

Now that we know the syntax, let's see our code and check what JNDI name it uses. Since our stateless EJB named CalculatorBean is deployed in a jboss-as-ejb-remote-app.jar (without any ear) and since we are looking up the org.jboss.as.quickstarts.ejb.remote.stateless.RemoteCalculator remote interface, our JNDI name will be:

```
ejb:/jboss-as-ejb-remote-app//CalculatorBean!org.jboss.as.quickstarts.ejb.remote.stateless.RemoteC
```

That's what the lookupRemoteStatelessCalculator() method in the above client code uses.

For the stateful EJB named CounterBean which is deployed in hte same jboss-as-ejb-remote-app.jar and which exposes the org.jboss.as.quickstarts.ejb.remote.stateful.RemoteCounter, the JNDI name will be:

```
ejb:/jboss-as-ejb-remote-app//CounterBean!org.jboss.as.quickstarts.ejb.remote.stateful.RemoteCount
```

That's what the lookupRemoteStatefulCounter() method in the above client code uses.

Now that we know of the JNDI name, let's take a look at the following piece of code in the lookupRemoteStatelessCalculator():

```
final Hashtable jndiProperties = new Hashtable();
jndiProperties.put(Context.URL_PKG_PREFIXES, "org.jboss.ejb.client.naming");
final Context context = new InitialContext(jndiProperties);
```

Here we are creating a JNDI InitialContext object by passing it some JNDI properties. The Context.URL_PKG_PREFIXES is set to org.jboss.ejb.client.naming. This is necessary because we should let the JNDI API know what handles the ejb: namespace that we use in our JNDI names for lookup. The "org.jboss.ejb.client.naming" has a URLContextFactory implementation which will be used by the JNDI APIs to parse and return an object for ejb: namespace lookups. You can either pass these properties to the constructor of the InitialContext class or have a jndi.properites file in the classpath of the client application, which (atleast) contains the following property:

```
java.naming.factory.url.pkgs=org.jboss.ejb.client.naming
```

So at this point, we have setup the InitialContext and also have the JNDI name ready to do the lookup. You can now do the lookup and the appropriate proxy which will be castable to the remote interface that you used as the fully qualified class name in the JNDI name, will be returned. Some of you might be wondering, how the JNDI implementation knew which server address to look, for your deployed EJBs. The answer is in AS7, the proxies returned via JNDI name lookup for ejb: namespace do not connect to the server unless an invocation on those proxies is done.

Now let's get to the point where we invoke on this returned proxy:

```
// Let's lookup the remote stateless calculator
        final RemoteCalculator statelessRemoteCalculator = lookupRemoteStatelessCalculator();
        System.out.println("Obtained a remote stateless calculator for invocation");
        // invoke on the remote calculator
        int a = 204;
        int b = 340;
        System.out.println("Adding " + a + " and " + b + " via the remote stateless calculator
deployed on the server");
        int sum = statelessRemoteCalculator.add(a, b);
```

We can see here that the proxy returned after the lookup is used to invoke the add(...) method of the bean. It's at this point that the JNDI implementation (which is backed by the EJB client API) needs to know the server details. So let's now get to the important part of setting up the EJB client context properties.

6.5.3 Setting up EJB client context properties

A EJB client context is a context which contains contextual information for carrying out remote invocations on EJBs. This is a WildFly-specific API. The EJB client context can be associated with multiple EJB receivers. Each EJB receiver is capable of handling invocations on different EJBs. For example, an EJB receiver "Foo" might be able to handle invocation on a bean identified by app-A/module-A/distinctinctName-A/Bar!RemoteBar, whereas a EJB receiver named "Blah" might be able to handle invocation on a bean identified by app-B/module-B/distinctName-B/BeanB!RemoteBean. Each such EJB receiver knows about what set of EJBs it can handle and each of the EJB receiver knows which server target to use for handling the invocations on the bean. For example, if you have a AS7 server at 10.20.30.40 IP address which has its remoting port opened at 4447 and if that's the server on which you deployed that CalculatorBean, then you can setup a EJB receiver which knows its target address is 10.20.30.40:4447. Such an EJB receiver will be capable enough to communicate to the server via the JBoss specific EJB remote client protocol (details of which will be explained in-depth in a separate chapter).

Now that we know what a EJB client context is and what a EJB receiver is, let's see how we can setup a client context with 1 EJB receiver which can connect to 10.20.30.40 IP address at port 4447. That EJB client context will then be used (internally) by the JNDI implementation to handle invocations on the bean proxy.

The client will have to place a jboss-ejb-client.properties file in the classpath of the application. The jboss-ejb-client.properties can contain the following properties:

```
endpoint.name=client-endpoint
remote.connectionprovider.create.options.org.xnio.Options.SSL_ENABLED=false

remote.connections=default

remote.connection.default.host=10.20.30.40
remote.connection.default.port = 8080
remote.connection.default.connect.options.org.xnio.Options.SASL_POLICY_NOANONYMOUS=false

remote.connection.default.username=appuser
remote.connection.default.password=apppassword
```

This file includes a reference to a default password. Be sure to change this as soon as possible.

The above properties file is just an example. The actual file that was used for this sample program is available here for reference jboss-ejb-client.properties

We'll see what each of it means.

First the endpoint.name property. We mentioned earlier that the EJB receivers will communicate with the server for EJB invocations. Internally, they use JBoss Remoting project to carry out the communication. The endpoint.name property represents the name that will be used to create the client side of the enpdoint. The endpoint.name property is optional and if not specified in the jboss-ejb-client.properties file, it will default to "config-based-ejb-client-endpoint" name.

Next is the remote.connectionprovider.create.options.<....> properties:

```
remote.connectionprovider.create.options.org.xnio.Options.SSL_ENABLED=false
```

The "remote.connectionprovider.create.options." property prefix can be used to pass the options that will be used while create the connection provider which will handle the "remote:" protocol. In this example we use the "remote.connectionprovider.create.options." property prefix to pass the "org.xnio.Options.SSL_ENABLED" property value as false. That property will then be used during the connection provider creation. Similarly other properties can be passed too, just append it to the "remote.connectionprovider.create.options." prefix

Next we'll see:

```
remote.connections=default
```

This is where you define the connections that you want to setup for communication with the remote server. The "remote.connections" property uses a comma separated value of connection "names". The connection names are just logical and are used grouping together the connection configuration properties later on in the properties file. The example above sets up a single remote connection named "default". There can be more than one connections that are configured. For example:

```
remote.connections=one, two
```

Here we are listing 2 connections named "one" and "two". Ultimately, each of the connections will map to a EJB receiver. So if you have 2 connections, that will setup 2 EJB receivers that will be added to the EJB client context. Each of these connections will be configured with the connection specific properties as follows:

```
remote.connection.default.host=10.20.30.40
remote.connection.default.port = 8080
remote.connection.default.connect.options.org.xnio.Options.SASL_POLICY_NOANONYMOUS=false
```

As you can see we are using the "remote.connection.<connection-name>." prefix for specifying the connection specific property. The connection name here is "default" and we are setting the "host" property of that connection to point to 10.20.30.40. Similarly we set the "port" for that connection to 4447.

By default WildFly uses 8080 as the remoting port. The EJB client API uses the http port, with the http-upgrade functionality, for communicating with the server for remote invocations, so that's the port we use in our client programs (unless the server is configured for some other http port)

```
remote.connection.default.username=appuser
remote.connection.default.password=apppassword
```

The given user/password must be set by using the command bin/add-user.sh (or.bat).
The user and password must be set because the security-realm is enabled for the subsystem remoting (see standalone*.xml or domain.xml) by default.
If you do not need the security for remoting you might remove the attribute security-realm in the configuration.

security-realm is enabled by default.

We then use the "remote.connection.<connection-name>.connect.options." property prefix to setup options that will be used during the connection creation.

Here's an example of setting up multiple connections with different properties for each of those:

```
remote.connectionprovider.create.options.org.xnio.Options.SSL_ENABLED=false

remote.connections=one, two

remote.connection.one.host=localhost
remote.connection.one.port=6999
remote.connection.one.connect.options.org.xnio.Options.SASL_POLICY_NOANONYMOUS=false

remote.connection.two.host=localhost
remote.connection.two.port=7999
remote.connection.two.connect.options.org.xnio.Options.SASL_POLICY_NOANONYMOUS=false
```

As you can see we setup 2 connections "one" and "two" which both point to "localhost" as the "host" but different ports. Each of these connections will internally be used to create the EJB receivers in the EJB client context.

So that's how the jboss-ejb-client.properties file can be setup and placed in the classpath.

Using a different file for setting up EJB client context

The EJB client code will by default look for jboss-ejb-client.properties in the classpath. However, you can specify a different file of your choice by setting the "jboss.ejb.client.properties.file.path" system property which points to a properties file on your filesystem, containing the client context configurations. An example for that would be "-Djboss.ejb.client.properties.file.path=/home/me/my-client/custom-jboss-ejb-client.properties"

Setting up the client classpath with the jars that are required to run the client application

A jboss-client jar is shipped in the distribution. It's available at WILDFLY_HOME/bin/client/jboss-client.jar. Place this jar in the classpath of your client application.

If you are using Maven to build the client application, then please follow the instructions in the WILDFLY_HOME/bin/client/README.txt to add this jar as a Maven dependency.

6.5.4 Summary

In the above examples, we saw what it takes to invoke a EJB from a remote client. To summarize:

- On the server side you need to deploy EJBs which expose the remote views.
- On the client side you need a client program which:
  - Has a jboss-ejb-client.properties in its classpath to setup the server connection information
  - Either has a jndi.properties to specify the java.naming.factory.url.pkgs property or passes that as a property to the InitialContext constructor
  - Setup the client classpath to include the jboss-client jar that's required for remote invocation of the EJBs. The location of the jar is mentioned above. You'll also need to have your application's bean interface jars and other jars that are required by your application, in the client classpath

6.6 EJB invocations from a remote server

The purpose of this chapter is to demonstrate how to lookup and invoke on EJBs deployed on an WildFly server instance **from another** WildFly server instance. This is different from invoking the EJBs from a remote standalone client

Let's call the server, from which the invocation happens to the EJB, as "Client Server" and the server on which the bean is deployed as the "Destination Server".

Note that this chapter deals with the case where the bean is deployed on the "Destination Server" but **not** on the "Client Server".

6.6.1 Application packaging

In this example, we'll consider a EJB which is packaged in a myejb.jar which is within a myapp.ear. Here's how it would look like:

```
myapp.ear
|
|---- myejb.jar
|        |
|        |---- <org.myapp.ejb.*> // EJB classes
```

Note that packaging itself isn't really important in the context of this article. You can deploy the EJBs in any standard way (.ear, .war or .jar).

6.6.2 Beans

In our example, we'll consider a simple stateless session bean which is as follows:

```
package org.myapp.ejb;

public interface Greeter {

    String greet(String user);
}
```

```
package org.myapp.ejb;

import javax.ejb.Remote;
import javax.ejb.Stateless;

@Stateless
@Remote (Greeter.class)
public class GreeterBean implements Greeter {

    @Override
    public String greet(String user) {
        return "Hello " + user + ", have a pleasant day!";
    }
}
```

6.6.3 Security

WildFly 8 is secure by default. What this means is that no communication can happen with an WildFly instance from a remote client (irrespective of whether it is a standalone client or another server instance) without passing the appropriate credentials. Remember that in this example, our "client server" will be communicating with the "destination server". So in order to allow this communication to happen successfully, we'll have to configure user credentials which we will be using during this communication. So let's start with the necessary configurations for this.

6.6.4 Configuring a user on the "Destination Server"

As a first step we'll configure a user on the destination server who will be allowed to access the destination server. We create the user using the `add-user` script that's available in the JBOSS_HOME/bin folder. In this example, we'll be configuring a `Application User` named `ejb` and with a password `test` in the `ApplicationRealm`. Running the `add-user` script is an interactive process and you will see questions/output as follows:

add-user

```
jpai@jpai-laptop:bin$ ./add-user.sh

What type of user do you wish to add?
 a) Management User (mgmt-users.properties)
 b) Application User (application-users.properties)
(a): b

Enter the details of the new user to add.
Realm (ApplicationRealm) :
Username : ejb
Password :
Re-enter Password :
What roles do you want this user to belong to? (Please enter a comma separated list, or leave
blank for none)\[  \]:
About to add user 'ejb' for realm 'ApplicationRealm'
Is this correct yes/no? yes
Added user 'ejb' to file
'/jboss-as-7.1.1.Final/standalone/configuration/application-users.properties'
Added user 'ejb' to file
'/jboss-as-7.1.1.Final/domain/configuration/application-users.properties'
Added user 'ejb' with roles to file
'/jboss-as-7.1.1.Final/standalone/configuration/application-roles.properties'
Added user 'ejb' with roles to file
'/jboss-as-7.1.1.Final/domain/configuration/application-roles.properties'
```

As you can see in the output above we have now configured a user on the destination server who'll be allowed to access this server. We'll use this user credentials later on in the client server for communicating with this server. The important bits to remember are the user we have created in this example is `ejb` and the password is `test.`

Note that you can use any username and password combination you want to.

You do **not** require the server to be started to add a user using the add-user script.

6.6.5 Start the "Destination Server"

As a next step towards running this example, we'll start the "Destination Server". In this example, we'll use the standalone server and use the *standalone-full.xml* configuration. The startup command will look like:

```
./standalone.sh -server-config=standalone-full.xml
```

Ensure that the server has started without any errors.

It's very important to note that if you are starting both the server instances on the same machine, then each of those server instances **must** have a unique `jboss.node.name` system property. You can do that by passing an appropriate value for `-Djboss.node.name` system property to the startup script:

```
./standalone.sh -server-config=standalone-full.xml -Djboss.node.name=<add appropriate
value here>
```

6.6.6 Deploying the application

The application (*myapp.ear* in our case) will be deployed to "Destination Server". The process of deploying the application is out of scope of this chapter. You can either use the Command Line Interface or the Admin console or any IDE or manually copy it to JBOSS_HOME/standalone/deployments folder (for standalone server). Just ensure that the application has been deployed successfully.

So far, we have built a EJB application and deployed it on the "Destination Server". Now let's move to the "Client Server" which acts as the client for the deployed EJBs on the "Destination Server".

6.6.7 Configuring the "Client Server" to point to the EJB remoting connector on the "Destination Server"

As a first step on the "Client Server", we need to let the server know about the "Destination Server"'s EJB remoting connector, over which it can communicate during the EJB invocations. To do that, we'll have to add a "*remote-outbound-connection*" to the remoting subsystem on the "Client Server". The "*remote-outbound-connection*" configuration indicates that a outbound connection will be created to a remote server instance from that server. The "*remote-outbound-connection*" will be backed by a "*outbound-socket-binding*" which will point to a remote host and a remote port (of the "Destination Server"). So let's see how we create these configurations.

6.6.8 Start the "Client Server"

In this example, we'll start the "Client Server" on the same machine as the "Destination Server". We have copied the entire server installation to a different folder and while starting the "Client Server" we'll use a port-offset (of 100 in this example) to avoid port conflicts:

```
./standalone.sh -server-config=standalone-full.xml -Djboss.socket.binding.port-offset=100
```

6.6.9 Create a security realm on the client server

Remember that we need to communicate with a secure destination server. In order to do that the client server has to pass the user credentials to the destination server. Earlier we created a user on the destination server who'll be allowed to communicate with that server. Now on the "client server" we'll create a security-realm which will be used to pass the user information.

In this example we'll use a security realm which stores a Base64 encoded password and then passes on that credentials when asked for. Earlier we created a user named `ejb` and password `test`. So our first task here would be to create the base64 encoded version of the password `test`. You can use any utility which generates you a base64 version for a string. I used this online site which generates the base64 encoded string. So for the `test` password, the base64 encoded version is `dGVzdA==`

While generating the base64 encoded string make sure that you don't have any trailing or leading spaces for the original password. That can lead to incorrect encoded versions being generated.

With new versions the add-user script will show the base64 password if you type 'y' if you've been ask

```
Is this new user going to be used for one AS process to connect to another AS process
e.g. slave domain controller?
```

Now that we have generated that base64 encoded password, let's use in the in the security realm that we are going to configure on the "client server". I'll first shutdown the client server and edit the standalone-full.xml file to add the following in the `<management>` section

Now let's create a "*security-realm*" for the base64 encoded password.

```
/core-service=management/security-realm=ejb-security-realm:add()
/core-service=management/security-realm=ejb-security-realm/server-identity=secret:add(value=dGVzdA
```

Notice that the CLI show the message *"process-state" => "reload-required"*, so you have to restart the server before you can use this change.

upon successful invocation of this command, the following configuration will be created in the *management* section:

standalone-full.xml

```
<management>
        <security-realms>
            ...
            <security-realm name="ejb-security-realm">
                <server-identities>
                    <secret value="dGVzdA==" />
                </server-identities>
            </security-realm>
        </security-realms>
...
```

As you can see I have created a security realm named "ejb-security-realm" (you can name it anything) with the base64 encoded password. So that completes the security realm configuration for the client server. Now let's move on to the next step.

6.6.10 Create a outbound-socket-binding on the "Client Server"

Let's first create a *outbound-socket-binding* which points the "Destination Server"'s host and port. We'll use the CLI to create this configuration:

```
/socket-binding-group=standard-sockets/remote-destination-outbound-socket-binding=remote-ejb:add(h
port=8080)
```

The above command will create a outbound-socket-binding named "*remote-ejb*" (we can name it anything) which points to "localhost" as the host and port 8080 as the destination port. Note that the host information should match the host/IP of the "Destination Server" (in this example we are running on the same machine so we use "localhost") and the port information should match the http-remoting connector port used by the EJB subsystem (by default it's 8080). When this command is run successfully, we'll see that the standalone-full.xml (the file which we used to start the server) was updated with the following outbound-socket-binding in the socket-binding-group:

```
<socket-binding-group name="standard-sockets" default-interface="public"
port-offset="${jboss.socket.binding.port-offset:0}">
        ...
        <outbound-socket-binding name="remote-ejb">
            <remote-destination host="localhost" port="8080"/>
        </outbound-socket-binding>
    </socket-binding-group>
```

6.6.11 Create a "remote-outbound-connection" which uses this newly created "outbound-socket-binding"

Now let's create a "*remote-outbound-connection*" which will use the newly created outbound-socket-binding (pointing to the EJB remoting connector of the "Destination Server"). We'll continue to use the CLI to create this configuration:

```
/subsystem=remoting/remote-outbound-connection=remote-ejb-connection:add(outbound-socket-binding-r
protocol=http-remoting, security-realm=ejb-security-realm, username=ejb)
```

The above command creates a remote-outbound-connection, named "*remote-ejb-connection*" (we can name it anything), in the remoting subsystem and uses the previously created "*remote-ejb*" outbound-socket-binding (notice the outbound-socket-binding-ref in that command) with the http-remoting protocol. Furthermore, we also set the security-realm attribute to point to the security-realm that we created in the previous step. Also notice that we have set the username attribute to use the user name who is allowed to communicate with the destination server.

What this step does is, it creates a outbound connection, on the client server, to the remote destination server and sets up the username to the user who allowed to communicate with that destination server and also sets up the security-realm to a pre-configured security-realm capable of passing along the user credentials (in this case the password). This way when a connection has to be established from the client server to the destination server, the connection creation logic will have the necessary security credentials to pass along and setup a successful secured connection.

Now let's run the following two operations to set some default connection creation options for the outbound connection:

```
/subsystem=remoting/remote-outbound-connection=remote-ejb-connection/property=SASL_POLICY_NOANONYM
```

```
/subsystem=remoting/remote-outbound-connection=remote-ejb-connection/property=SSL_ENABLED:add(valu
```

Ultimately, upon successful invocation of this command, the following configuration will be created in the remoting subsystem:

```
<subsystem xmlns="urn:jboss:domain:remoting:1.1">
....
            <outbound-connections>
                <remote-outbound-connection name="remote-ejb-connection"
outbound-socket-binding-ref="remote-ejb" protocol="http-remoting"
security-realm="ejb-security-realm" username="ejb">
                    <properties>
                        <property name="SASL_POLICY_NOANONYMOUS" value="false"/>
                        <property name="SSL_ENABLED" value="false"/>
                    </properties>
                </remote-outbound-connection>
            </outbound-connections>
        </subsystem>
```

From a server configuration point of view, that's all we need on the "Client Server". Our next step is to deploy an application on the "Client Server" which will invoke on the bean deployed on the "Destination Server".

6.6.12 Packaging the client application on the "Client Server"

Like on the "Destination Server", we'll use .ear packaging for the client application too. But like previously mentioned, that's not mandatory. You can even use a .war or .jar deployments. Here's how our client application packaging will look like:

```
client-app.ear
|
|--- META-INF
|        |
|        |--- jboss-ejb-client.xml
|
|--- web.war
|        |
|        |--- WEB-INF/classes
|        |        |
|        |        |---- <org.myapp.FooServlet> // classes in the web app
```

In the client application we'll use a servlet which invokes on the bean deployed on the "Destination Server". We can even invoke the bean on the "Destination Server" from a EJB on the "Client Server". The code remains the same (JNDI lookup, followed by invocation on the proxy). The important part to notice in this client application is the file *jboss-ejb-client.xml* which is packaged in the META-INF folder of a top level deployment (in this case our client-app.ear). This *jboss-ejb-client.xml* contains the EJB client configurations which will be used during the EJB invocations for finding the appropriate destinations (also known as, EJB receivers). The contents of the jboss-ejb-client.xml are explained next.

If your application is deployed as a top level .war deployment, then the jboss-ejb-client.xml is expected to be placed in .war/WEB-INF/ folder (i.e. the same location where you place any web.xml file).

6.6.13 Contents on jboss-ejb-client.xml

The jboss-ejb-client.xml will look like:

```
<jboss-ejb-client xmlns="urn:jboss:ejb-client:1.0">
    <client-context>
        <ejb-receivers>
            <remoting-ejb-receiver outbound-connection-ref="remote-ejb-connection"/>
        </ejb-receivers>
    </client-context>
</jboss-ejb-client>
```

You'll notice that we have configured the EJB client context (for this application) to use a remoting-ejb-receiver which points to our earlier created "*remote-outbound-connection*" named "*remote-ejb-connection*". This links the EJB client context to use the "*remote-ejb-connection*" which ultimately points to the EJB remoting connector on the "Destination Server".

6.6.14 Deploy the client application

Let's deploy the client application on the "Client Server". The process of deploying the application is out of scope, of this chapter. You can use either the CLI or the admin console or a IDE or deploy manually to JBOSS_HOME/standalone/deployments folder. Just ensure that the application is deployed successfully.

6.6.15 Client code invoking the bean

We mentioned that we'll be using a servlet to invoke on the bean, but the code to invoke the bean isn't servlet specific and can be used in other components (like EJB) too. So let's see how it looks like:

```
import javax.naming.Context;
import java.util.Hashtable;
import javax.naming.InitialContext;

...
public void invokeOnBean() {
        try {
            final Hashtable props = new Hashtable();
            // setup the ejb: namespace URL factory
            props.put(Context.URL_PKG_PREFIXES, "org.jboss.ejb.client.naming");
            // create the InitialContext
            final Context context = new javax.naming.InitialContext(props);

            // Lookup the Greeter bean using the ejb: namespace syntax which is explained here
https://docs.jboss.org/author/display/AS71/EJB+invocations+from+a+remote+client+using+JNDI
            final Greeter bean = (Greeter) context.lookup("ejb:" + "myapp" + "/" + "myejb" + "/"
+ "" + "/" + "GreeterBean" + "!" + org.myapp.ejb.Greeter.class.getName());

   // invoke on the bean
   final String greeting = bean.greet("Tom");

            System.out.println("Received greeting: " + greeting);

        } catch (Exception e) {
            throw new RuntimeException(e);
        }
}
```

That's it! The above code will invoke on the bean deployed on the "Destination Server" and return the result.

6.7 Remote EJB invocations via JNDI - Which approach to use?

Couldn't find a page to include called: Remote EJB invocations via JNDI - EJB client API or remote-naming project?

6.8 JBoss EJB 3 reference guide

This chapter details the extensions that are available when developing Enterprise Java Beans [tm] on WildFly 8.

Currently there is no support for configuring the extensions using an implementation specific descriptor file.

6.8.1 Resource Adapter for Message Driven Beans

Each Message Driven Bean must be connected to a resource adapter.

Specification of Resource Adapter using Metadata Annotations

The `ResourceAdapter` annotation is used to specify the resource adapter with which the MDB should connect.

The `value` of the annotation is the name of the deployment unit containing the resource adapter. For example `jms-ra.rar`.

For example:

```
@MessageDriven(messageListenerInterface = PostmanPat.class)
@ResourceAdapter("ejb3-rar.rar")
```

6.8.2 as Principal

Whenever a run-as role is specified for a given method invocation the default anonymous principal is used as the caller principal. This principal can be overridden by specifying a run-as principal.

Specification of Run-as Principal using Metadata Annotations

The `RunAsPrincipal` annotation is used to specify the run-as principal to use for a given method invocation.

The `value` of the annotation specifies the name of the principal to use. The actual type of the principal is undefined and should not be relied upon.

Using this annotation without specifying a run-as role is considered an error.

For example:

```
@RunAs("admin")
@RunAsPrincipal("MyBean")
```

6.8.3 Security Domain

Each Enterprise Java Bean [tm] can be associated with a security domain. Only when an EJB is associated with a security domain will authentication and authorization be enforced.

Specification of Security Domain using Metadata Annotations

The `SecurityDomain` annotation is used to specify the security domain to associate with the EJB.

The `value` of the annotation is the name of the security domain to be used.

For example:

```
@SecurityDomain("other")
```

6.8.4 Transaction Timeout

For any newly started transaction a transaction timeout can be specified in seconds.

When a transaction timeout of `0` is used, then the actual transaction timeout will default to the domain configured default.
TODO: add link to tx subsystem

Although this is only applicable when using transaction attribute `REQUIRED` or `REQUIRES_NEW` the application server will not detect invalid setups.

New Transactions

Take care that even when transaction attribute `REQUIRED` is specified, the timeout will only be applicable if a **new** transaction is started.

Specification of Transaction Timeout with Metadata Annotations

The `TransactionTimeout` annotation is used to specify the transaction timeout for a given method.

The `value` of the annotation is the timeout used in the given `unit` granularity. It must be a positive integer or 0. Whenever 0 is specified the default domain configured timeout is used.

The `unit` specifies the granularity of the `value`. The actual value used is converted to seconds. Specifying a granularity lower than `SECONDS` is considered an error, even when the computed value will result in an even amount of seconds.

For example:@TransactionTimeout(value = 10, unit = TimeUnit.SECONDS)

Specification of Transaction Timeout in the Deployment Descriptor

The `trans-timeout` element is used to define the transaction timeout for business, home, component, and message-listener interface methods; no-interface view methods; web service endpoint methods; and timeout callback methods.

The `trans-timeout` element resides in the `urn:trans-timeout` namespace and is part of the standard `container-transaction` element as defined in the jboss namespace.

For the rules when a `container-transaction` is applicable please refer to EJB 3.1 FR 13.3.7.2.1.

Example of trans-timeout

jboss-ejb3.xml

```
<jboss:ejb-jar xmlns:jboss="http://www.jboss.com/xml/ns/javaee"
               xmlns="http://java.sun.com/xml/ns/javaee"
               xmlns:tx="urn:trans-timeout"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee
http://www.jboss.org/j2ee/schema/jboss-ejb3-2_0.xsd
http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd
urn:trans-timeout http://www.jboss.org/j2ee/schema/trans-timeout-1_0.xsd"
               version="3.1"
               impl-version="2.0">
    <assembly-descriptor>
        <container-transaction>
            <method>
                <ejb-name>BeanWithTimeoutValue</ejb-name>
                <method-name>*</method-name>
                <method-intf>Local</method-intf>
            </method>
            <tx:trans-timeout>
                <tx:timeout>10</tx:timeout>
                <tx:unit>Seconds</tx:unit>
            </tx:trans-timeout>
        </container-transaction>
    </assembly-descriptor>
</jboss:ejb-jar>
```

6.8.5 Timer service

The service is responsible to call the registered timeout methods of the different session beans.

A persistent timer will be identified by the name of the EAR, the name of the sub-deployment JAR and the Bean's name.
If one of those names are changed (e.g. EAR name contain a version) the timer entry became orphaned and the timer event will not longer be fired.

Single event timer

The timer is will be started once at the specified time.

In case of a server restart the timeout method of a persistent timer will only be called directly if the specified time is elapsed.
If the timer is not persistent (since EJB3.1 see 18.2.3) it will be not longer available if JBoss is restarted or the application is redeployed.

Recurring timer

The timer will be started at the specified first occurrence and after that point at each time if the interval is elapsed.
If the timer will be started during the last execution is not finished the execution will be suppressed with a warning to avoid concurrent execution.

In case of server downtime for a persistent timer, the timeout method will be called only once if one, or more than one, interval is elapsed.
If the timer is not persistent (since EJB3.1 see 18.2.3) it will not longer be active after the server is restarted or the application is redeployed.

Calendar timer

The timer will be started if the schedule expression match. It will be automatically deactivated and removed if there will be no next expiration possible, i.e. If you set a specific year.

For example:

```
@Schedule( ... dayOfMonth="1", month="1", year="2012")
// start once at 01-01-2012 00:00:00
```

Programmatic calendar timer

If the timer is persistent it will be fetched at server start and the missed timeouts are called concurrent.
If a persistent timer contains an end date it will be executed once nevertheless how many times the execution was missed. Also a retry will be suppressed if the timeout method throw an Exception.
In case of such expired timer access to the given Timer object might throw a NoMoreTimeoutExcption or NoSuchObjectException.

If the timer is non persistent it will not longer be active after the server is restarted or the application is redeployed.

TODO: clarify whether this should happen concurrently/blocked or even fired only once like a recurring timer!

Annotated calendar timer

If the timer is non persistent it will not activated for missed events during the server is down. In case of server start the timer is scheduled based on the @Schedule annotation.

If the timer is persistent (default if not deactivated by annotation) all missed events are fetched at server start and the annotated timeout method is called concurrent.

TODO: clarify whether this should happen concurrently/blocked or even fired only once like a recurring timer!

6.9 JPA reference guide

6.9.1 Introduction

The WildFly JPA subsystem implements the JPA 2.1 container-managed requirements. Deploys the persistence unit definitions, the persistence unit/context annotations and persistence unit/context references in the deployment descriptor. JPA Applications use the Hibernate (version 5.1) persistence provider, which is included with WildFly. The JPA subsystem uses the standard SPI (javax.persistence.spi.PersistenceProvider) to access the Hibernate persistence provider and some additional extensions as well.

During application deployment, JPA use is detected (e.g. persistence.xml or @PersistenceContext/Unit annotations) and injects Hibernate dependencies into the application deployment. This makes it easy to deploy JPA applications.

In the remainder of this documentation, ”entity manager” refers to an instance of the *javax.persistence.EntityManager* class. Javadoc for the JPA interfaces and JPA 2.1 specification.

The index of the Hibernate documentation is at http://hibernate.org/orm/documentation/5.1/.

6.9.2 Update your Persistence.xml for Hibernate 5.1

The persistence provider class name in Hibernate 4.3.0 (and greater) is **org.hibernate.jpa.HibernatePersistenceProvider**.

Instead of specifying:

<provider>org.hibernate.ejb.HibernatePersistence</provider>

Switch to:

<provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>

Or remove the persistence provider class name from your persistence.xml (so the default provider will be used).

6.9.3 Entity manager

The entity manager (javax.persistence.EntityManager class) is similar to the Hibernate Session class; applications use it to create/read/update/delete data (and related operations). Applications can use application-managed or container-managed entity managers. Keep in mind that the entity manager is not thread safe, don't share the same entity manager instance with multiple threads.

Internally, the entity manager, has a persistence context for managing entities. You can think of the persistence context as being closely associated with the entity manager.

6.9.4 Container-managed entity manager

When you inject a container-managed entity managers into an application variable, it is treated like an (EE container controlled) Java proxy object, that will be associated with an underlying EntityManager instance, for each started JTA transaction and is flushed/closed when the JTA transaction commits. Such that when your application code invokes EntityManager.anyMethod(), the current JTA transaction is searched (using persistence unit name as key) for the underlying EntityManager instance, if not found, a new EntityManager instance is created and associated with the current JTA transaction, to be reused for the next EntityManager invocation. Use the @PersistenceContext annotation, to inject a container-managed entity manager into a javax.persistence.EntityManager variable.

6.9.5 Application-managed entity manager

An application-managed entity manager is kept around until the application closes it. The scope of the application-managed entity manager is from when the application creates it and lasts until the application closes it. Use the *@PersistenceUnit* annotation, to inject a persistence unit into a *javax.persistence.EntityManagerFactory variable*. The EntityManagerFactory can return an application-managed entity manager.

6.9.6 Persistence Context

The JPA persistence context contains the entities managed by the entity manager (via the JPA persistence provider). The underlying entity manager maintains the persistence context. The persistence context acts like a first level (transactional) cache for interacting with the datasource. Loaded entities are placed into the persistence context before being returned to the application. Entities changes are also placed into the persistence context (to be saved in the database when the transaction commits).

6.9.7 Transaction-scoped Persistence Context

The transaction-scoped persistence context coordinates with the (active) JTA transaction. When the transaction commits, the persistence context is flushed to the datasource (entity objects are detached but may still be referenced by application code). All entity changes that are expected to be saved to the datasource, must be made during a transaction. Entities read outside of a transaction will be detached when the entity manager invocation completes. Example transaction-scoped persistence context is below.

```
@Stateful  // will use container managed transactions
public class CustomerManager {
  @PersistenceContext(unitName = "customerPU") // default type is
PersistenceContextType.TRANSACTION
  EntityManager em;
  public customer createCustomer(String name, String address) {
    Customer customer = new Customer(name, address);
    em.persist(customer);  // persist new Customer when JTA transaction completes (when method
ends).
                           // internally:
                           //    1. Look for existing "customerPU" persistence context in active
JTA transaction and use if found.
                           //    2. Else create new "customerPU" persistence context (e.g.
instance of org.hibernate.ejb.HibernatePersistence)
                           //       and put in current active JTA transaction.
    return customer;       // return Customer entity (will be detached from the persistence
context when caller gets control)
  }  // Transaction.commit will be called, Customer entity will be persisted to the database and
"customerPU" persistence context closed
```

6.9.8 Extended Persistence Context

The (ee container managed) extended persistence context can span multiple transactions and allows data modifications to be queued up (like a shopping cart), without an active JTA transaction (to be applied during the next JTA TX). The Container-managed extended persistence context can only be injected into a stateful session bean. You can also think of the extended persistence context, as being an entity manager.

```
@PersistenceContext(type = PersistenceContextType.EXTENDED, unitName = "inventoryPU")
EntityManager em;
```

Extended Persistence Context Inheritance

```
JPA 2.0 specification section 7.6.2.1

If a stateful session bean instantiates a stateful session bean (executing in the same EJB
container instance) which also has such an extended persistence context, the extended
persistence context of the first stateful session bean is inherited by the second stateful
session bean and bound to it, and this rule recursively applies—independently of whether
transactions are active or not at the point of the creation of the stateful session beans.
```

By default, the current stateful session bean being created, will (**deeply**) inherit the extended persistence context from any stateful session bean executing in the current Java thread. The **deep** inheritance of extended persistence context includes walking multiple levels up the stateful bean call stack (inheriting from parent beans). The **deep** inheritance of extended persistence context includes sibling beans. For example, parentA references child beans beanBwithXPC & beanCwithXPC. Even though parentA doesn't have an extended persistence context, beanBwithXPC & beanCwithXPC will share the same extended persistence context.

Some other EE application servers, use **shallow** inheritance, where stateful session bean only inherit from the parent stateful session bean (if there is a parent bean). Sibling beans do not share the same extended persistence context unless their (common) parent bean also has the same extended persistence context.

Applications can include a (top-level) **jboss-all.xml** deployment descriptor that specifies either the (default) **DEEP** extended persistence context inheritance or **SHALLOW**.

The WF/docs/schema/jboss-jpa_1_0.xsd describes the **jboss-jpa** deployment descriptor that may be included in the **jboss-all.xml**. Below is an example of using **SHALLOW** extended persistence context inheritance:

```
<jboss>
   <jboss-jpa xmlns="http://www.jboss.com/xml/ns/javaee">
   <extended-persistence inheritance="SHALLOW"/>
   </jboss-jpa>
</jboss>
```

Below is an example of using **DEEP** extended persistence inheritance:

```
<jboss>
   <jboss-jpa xmlns="http://www.jboss.com/xml/ns/javaee">
   <extended-persistence inheritance="DEEP"/>
   </jboss-jpa>
</jboss>
```

The AS console/cli can change the **default** extended persistence context setting (DEEP or SHALLOW). The following cli commands will read the current JPA settings and enable SHALLOW extended persistence context inheritance for applications that do not include the **jboss-jpa** deployment descriptor:

> *./jboss-cli.sh*
> *cd subsystem=jpa*
> *:read-resource*
> *:write-attribute(name=default-extended-persistence-inheritance,value="SHALLOW")*

6.9.9 Entities

JPA allows use of your (pojo) plain old Java class to represent a database table row.

```
@PersistenceContext EntityManager em;
Integer bomPk = getIndexKeyValue();
BillOfMaterials bom = em.find(BillOfMaterials.class, bomPk); // read existing table row into
BillOfMaterials class

BillOfMaterials createdBom = new BillOfMaterials("...");     // create new entity
em.persist(createdBom);  // createdBom is now managed and will be saved to database when the
current JTA transaction completes
```

The entity lifecycle is managed by the underlying persistence provider.

- New (transient): an entity is new if it has just been instantiated using the new operator, and it is not associated with a persistence context. It has no persistent representation in the database and no identifier value has been assigned.
- Managed (persistent): a managed entity instance is an instance with a persistent identity that is currently associated with a persistence context.
- Detached: the entity instance is an instance with a persistent identity that is no longer associated with a persistence context, usually because the persistence context was closed or the instance was evicted from the context.
- Removed: a removed entity instance is an instance with a persistent identity, associated with a persistence context, but scheduled for removal from the database.

6.9.10 Deployment

The persistence.xml contains the persistence unit configuration (e.g. datasource name) and as described in the JPA 2.0 spec (section 8.2), the jar file or directory whose META-INF directory contains the persistence.xml file is termed the root of the persistence unit. In Java EE environments, the root of a persistence unit must be one of the following (quoted directly from the JPA 2.0 specification):

"

- an EJB-JAR file
- the WEB-INF/classes directory of a WAR file
- a jar file in the WEB-INF/lib directory of a WAR file
- a jar file in the EAR library directory
- an application client jar file

The persistence.xml can specify either a JTA datasource or a non-JTA datasource. The JTA datasource is expected to be used within the EE environment (even when reading data without an active transaction). If a datasource is not specified, the default-datasource will instead be used (must be configured).

NOTE: Java Persistence 1.0 supported use of a jar file in the root of the EAR as the root of a persistence unit. This use is no longer supported. Portable applications should use the EAR library directory for this case instead.

"

Question: Can you have a EAR/META-INF/persistence.xml?

Answer: No, the above may deploy but it could include other archives also in the EAR, so you may have deployment issues for other reasons. Better to put the persistence.xml in an EAR/lib/somePuJar.jar.

6.9.11 Troubleshooting

The **org.jboss.as.jpa** logging can be enabled to get the following information:

- INFO - when persistence.xml has been parsed, starting of persistence unit service (per deployed persistence.xml), stopping of persistence unit service
- DEBUG - informs about entity managers being injected, creating/reusing transaction scoped entity manager for active transaction
- TRACE - shows how long each entity manager operation took in milliseconds, application searches for a persistence unit, parsing of persistence.xml

To enable TRACE, open the as/standalone/configuration/standalone.xml (or as/domain/configuration/domain.xml) file. Search for **<subsystem xmlns="urn:jboss:domain:logging:1.0">** and add the **org.jboss.as.jpa** category. You need to change the console-handler level from **INFO** to **TRACE**.

```
<subsystem xmlns="urn:jboss:domain:logging:1.0">
     <console-handler name="CONSOLE">
      <level name="TRACE" />
      ...
     </console-handler>

     </periodic-rotating-file-handler>
     <logger category="com.arjuna">
        <level name="WARN" />
     </logger>

     <logger category="org.jboss.as.jpa">
        <level name="TRACE" />
     </logger>

     <logger category="org.apache.tomcat.util.modeler">
        <level name="WARN" />
     </logger>
     ...
```

To see what is going on at the JDBC level, enable **jboss.jdbc.spy** TRACE and add spy="true" to the datasource.

```
<datasource jndi-name="java:jboss/datasources/..." pool-name="..." enabled="true" spy="true">
<logger category="jboss.jdbc.spy">
  <level name="TRACE"/>
</logger>
```

To troubleshoot issues with the Hibernate second level cache, try enabling trace for **org.hibernate.SQL + org.hibernate.cache.infinispan + org.infinispan:**

```
<subsystem xmlns="urn:jboss:domain:logging:1.0">
     <console-handler name="CONSOLE">
      <level name="TRACE" />
      ...
     </console-handler>

     </periodic-rotating-file-handler>
     <logger category="com.arjuna">
        <level name="WARN" />
     </logger>

     <logger category="org.hibernate.SQL">
        <level name="TRACE" />
     </logger>

     <logger category="org.hibernate">
        <level name="TRACE" />
     </logger>
      <logger category="org.infinispan">
        <level name="TRACE" />
     </logger>

     <logger category="org.apache.tomcat.util.modeler">
        <level name="WARN" />
     </logger>
     ...
```

6.9.12 Using the Infinispan second level cache

To enable the second level cache with Hibernate 5.1, just set the **hibernate.cache.use_second_level_cache** property to true, as is done in the following example (also set the shared-cache-mode accordingly). By default the application server uses Infinispan as the cache provider for **JPA applications**, so you don't need specify anything on top of that. The Infinispan version that is included in WildFly is expected to work with the Hibernate version that is included with WildFly. Example persistence.xml settings:

```
<?xml version="1.0" encoding="UTF-8"?><persistence
xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">
<persistence-unit name="2lc_example_pu">
   <description>example of enabling the second level cache.</description>
   <jta-data-source>java:jboss/datasources/mydatasource</jta-data-source>
   <shared-cache-mode>ENABLE_SELECTIVE</shared-cache-mode>
   <properties>
      <property name="hibernate.cache.use_second_level_cache" value="true"/>
   </properties>
</persistence-unit>
</persistence>
```

Here is an example of enabling the second level cache for a Hibernate native API hibernate.cfg.xml file:

```
<property name="hibernate.cache.region.factory_class"
value="org.jboss.as.jpa.hibernate5.infinispan.InfinispanRegionFactory"/>
<property name="hibernate.cache.infinispan.cachemanager"
value="java:jboss/infinispan/container/hibernate"/>
<property name="hibernate.transaction.manager_lookup_class"
value="org.hibernate.transaction.JBossTransactionManagerLookup"/>
<property name="hibernate.cache.use_second_level_cache" value="true"/>
```

The Hibernate native API application will also need a MANIFEST.MF:

```
Dependencies: org.infinispan,org.hibernate
```

Infinispan Hibernate/JPA second level cache provider documentation contains advanced configuration information but you should bear in mind that when Hibernate runs within WildFly 8, some of those configuration options, such as region factory, are not needed. Moreover, the application server providers you with option of selecting a different cache container for Infinispan via **hibernate.cache.infinispan.container** persistence property. To reiterate, this property is not mandatory and a default container is already deployed for by the application server to host the second level cache.

Here is an example of what the Hibernate cache settings may currently be in your standalone.xml:

```
<cache-container name="hibernate" default-cache="local-query" module="org.hibernate.infinispan">
    <local-cache name="entity">
        <transaction mode="NON_XA"/>
        <eviction strategy="LRU" max-entries="10000"/>
        <expiration max-idle="100000"/>
    </local-cache>
    <local-cache name="local-query">
        <eviction strategy="LRU" max-entries="10000"/>
        <expiration max-idle="100000"/>
    </local-cache>
    <local-cache name="timestamps"/>
</cache-container>
```

Below is an example of customizing the "entity", "immutable-entity", "local-query", "pending-puts", "timestamps" cache configuration may look like:

```
<cache-container name="hibernate" module="org.hibernate.infinispan"
default-cache="immutable-entity">
    <local-cache name="entity">
        <transaction mode="NONE"/>
        <eviction max-entries="-1"/>
        <expiration max-idle="120000"/>
    </local-cache>
    <local-cache name="immutable-entity">
        <transaction mode="NONE"/>
        <eviction max-entries="-1"/>
        <expiration max-idle="120000"/>
    </local-cache>
    <local-cache name="local-query">
        <eviction max-entries="-1"/>
        <expiration max-idle="300000"/>
    </local-cache>
    <local-cache name="pending-puts">
        <transaction mode="NONE"/>
        <eviction strategy="NONE"/>
        <expiration max-idle="60000"/>
    </local-cache>
    <local-cache name="timestamps">
        <transaction mode="NONE"/>
        <eviction strategy="NONE"/>
    </local-cache>
</cache-container>
```

Persistence.xml to use the above custom settings:

```
<properties>
    <property name="hibernate.cache.use_second_level_cache" value="true"/>
    <property name="hibernate.cache.use_query_cache" value="true"/>
    <property name="hibernate.cache.infinispan.immutable-entity.cfg" value="immutable-entity"/>
    <property name="hibernate.cache.infinispan.timestamps.cfg" value="timestamps"/>
    <property name="hibernate.cache.infinispan.pending-puts.cfg" value="pending-puts"/>
</properties>
```

6.9.13 Replacing the current Hibernate 5.x jars with a newer version

Just update the current wildfly/modules/system/layers/base/org/hibernate/main folder to contain the newer version (after stopping your WildFly server instance).

1. Delete *.index files in wildfly/modules/system/layers/base/org/hibernate/main and wildfly/modules/system/layers/base/org/hibernate/envers/main folders.
2. Backup the current contents of wildfly/modules/system/layers/base/org/hibernate in case you make a mistake.
3. Remove the older jars and copy new Hibernate jars into wildfly/modules/system/layers/base/org/hibernate/main + wildfly/modules/system/layers/base/org/hibernate/envers/main.
4. Update the wildfly/modules/system/layers/base/org/hibernate/main/module.xml + wildfly/modules/system/layers/base/org/hibernate/envers/main/module.xml to name the jars that you copied in.
5. Also update the hibernate-infinispan jars in wildfly/modules/system/layers/base/org/hibernate/infinispan.

6.9.14 Using Hibernate Search

WildFly includes Hibernate Search. If you want to use the bundled version of Hibernate Search - which requires to use the default Hibernate ORM 5.1 persistence provider - this will be automatically enabled. Having this enabled means that, provided your application includes any entity which is annotated with **org.hibernate.search.annotations.Indexed**, the module **org.hibernate.search.orm:main** will be made available to your deployment; this will also include the required version of Apache Lucene.

If you do not want this module to be exposed to your deployment, set the persistence property **wildfly.jpa.hibernate.search.module** to either **none** to not automatically inject any Hibernate Search module, or to any other module identifier to inject a different module.
For example you could set **wildfly.jpa.hibernate.search.module=org.hibernate.search.orm:5.4.0.Alpha1** to use the experimental version 5.4.0.Alpha1 instead of the provided module; in this case you'll have to download and add the custom modules to the application server as other versions are not included.
When setting **wildfly.jpa.hibernate.search.module=none** you might also opt to include Hibernate Search and its dependencies within your application but we highly recommend the modules approach.

6.9.15 Packaging the Hibernate JPA persistence provider with your application

WildFly allows the packaging of Hibernate persistence provider jars with the application. The JPA deployer will detect the presence of a persistence provider in the application and **jboss.as.jpa.providerModule** needs to be set to **application**.<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">
<persistence-unit name="myOwnORMVersion_pu">
<description>Hibernate Persistence Unit.</description>
<jta-data-source>java:jboss/datasources/PlannerDS</jta-data-source>
<properties>
   <property name="jboss.as.jpa.providerModule" value="application" />
</properties>
</persistence-unit>
</persistence>

6.9.16 Migrating from OpenJPA

You need to copy the OpenJPA jars (e.g. openjpa-all.jar serp.jar) into the WildFly modules/org/apache/openjpa/main folder and update modules/org/apache/openjpa/main/module.xml to include the same jar file names that you copied in. This will help you get your application that depends on OpenJPA, to deploy on WildFly.

```
<module xmlns="urn:jboss:module:1.1" name="org.apache.openjpa">
    <resources>
        <resource-root path="jipijapa-openjpa-1.0.1.Final.jar"/>
        <resource-root path="openjpa-all.jar">
           <filter>
              <exclude path="javax/**" />
           </filter>
        </resource-root>
        <resource-root path="serp.jar"/>
    </resources>

    <dependencies>
        <module name="javax.api"/>
        <module name="javax.annotation.api"/>
        <module name="javax.enterprise.api"/>
        <module name="javax.persistence.api"/>
        <module name="javax.transaction.api"/>
        <module name="javax.validation.api"/>
        <module name="javax.xml.bind.api"/>
        <module name="org.apache.commons.collections"/>
        <module name="org.apache.commons.lang"/>
        <module name="org.jboss.as.jpa.spi"/>
        <module name="org.jboss.logging"/>
        <module name="org.jboss.vfs"/>
        <module name="org.jboss.jandex"/>
    </dependencies>
</module>
```

6.9.17 Migrating from EclipseLink

You need to copy the EclipseLink jar (e.g. eclipselink-2.6.0.jar or eclipselink.jar as in the example below) into the WildFly modules/org/eclipse/persistence/main folder and update modules/org/eclipse/persistence/main/module.xml to include the EclipseLink jar (take care to use the jar name that you copied in). If you happen to leave the EclipseLink version number in the jar name, the module.xml should reflect that. This will help you get your application that depends on EclipseLink, to deploy on WildFly.

```
<module xmlns="urn:jboss:module:1.1" name="org.eclipse.persistence">
    <resources>
        <resource-root path="jipijapa-eclipselink-10.0.0.Final.jar"/>
        <resource-root path="eclipselink.jar">           <filter>
              <exclude path="javax/**" />
           </filter>
        </resource-root>
    </resources>

    <dependencies>
        <module name="asm.asm"/>
        <module name="javax.api"/>
        <module name="javax.annotation.api"/>
        <module name="javax.enterprise.api"/>
        <module name="javax.persistence.api"/>
        <module name="javax.transaction.api"/>
        <module name="javax.validation.api"/>
        <module name="javax.xml.bind.api"/>
        <module name="org.antlr"/>
        <module name="org.apache.commons.collections"/>
        <module name="org.dom4j"/>
        <module name="org.jboss.as.jpa.spi"/>
        <module name="org.jboss.logging"/>
        <module name="org.jboss.vfs"/>
    </dependencies>
</module>
```

As a workaround for issueid=414974, set (WildFly) system property "eclipselink.archive.factory" to value "org.jipijapa.eclipselink.JBossArchiveFactoryImpl" via jboss-cli.sh command (WildFly server needs to be running when this command is issued):

```
jboss-cli.sh --connect
'/system-property=eclipselink.archive.factory:add(value=org.jipijapa.eclipselink.JBossArchiveFacto
```

. The following shows what the standalone.xml (or your WildFly configuration you are using) file might look like after updating the system properties:

```
<system-properties>
   ...
   <property name="eclipselink.archive.factory"
value="org.jipijapa.eclipselink.JBossArchiveFactoryImpl"/>
</system-properties>
```

You should then be able to deploy applications with persistence.xml that include;

```
<provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
```

Also refer to page how to use EclipseLink with WildFly guide here.

6.9.18 Migrating from DataNucleus

Read the how to use DataNucleus with WildFly guide here.

6.9.19 Native Hibernate use

Applications that use the Hibernate API directly, are referred to here as native Hibernate applications. Native Hibernate applications, can choose to use the Hibernate jars included with WildFly or they can package their own copy of the Hibernate jars. Applications that utilize JPA will automatically have the Hibernate classes injected onto the application deployment classpath. Meaning that JPA applications, should expect to use the Hibernate jars included in WildFly.

Example MANIFEST.MF entry to add dependency for Hibernate native applications:

```
Manifest-Version: 1.0
...
Dependencies: org.hibernate
```

If you use the Hibernate native api in your application and also use the JPA api to access the same entities (from the same Hibernate session/EntityManager), you could get surprising results (e.g. HibernateSession.saveOrUpdate(entity) is different than EntityManager.merge(entity). Each entity should be managed by either Hibernate native API or JPA code.

6.9.20 Injection of Hibernate Session and SessionFactoryInjection of Hibernate Session and SessionFactory

You can inject a org.hibernate.Session and org.hibernate.SessionFactory directly, just as you can do with EntityManagers and EntityManagerFactorys.

```
import org.hibernate.Session;
import org.hibernate.SessionFactory;
@Stateful public class MyStatefulBean ... {
   @PersistenceContext(unitName="crm") Session session1;
   @PersistenceContext(unitName="crm2", type=EXTENDED) Session extendedpc;
   @PersistenceUnit(unitName="crm") SessionFactory factory;
}
```

6.9.21 Hibernate properties

WildFly automatically sets the following Hibernate (5.x) properties (if not already set in persistence unit definition):

Property	Purpose
hibernate.id.new_generator_mappings =true	New applications should let th default to true, older applicatic with existing data might need set to false (see note below). really depends on whether yo application uses the @GeneratedValue(AUTO) wh will generates new key values newly created entities. The application can override this value (in the persistence.xml)
hibernate.transaction.jta.platform= instance of org.hibernate.service.jta.platform.spi.JtaPlatform interface	The transaction manager, use transaction and transaction synchronization registry is passed into Hibernate via this class.
hibernate.ejb.resource_scanner = instance of org.hibernate.ejb.packaging.Scanner interface	Instance of entity scanning cla is passed in that knows how t use the AS annotation indexe (for faster deployment).
hibernate.transaction.manager_lookup_class	This property is removed if fou in the persistence.xml (could conflict with JtaPlatform)
hibernate.session_factory_name = qualified persistence unit name	Is set to the application name persistence unit name (application can specify a different value but it needs to unique across all application deployments on the AS instance).
hibernate.session_factory_name_is_jndi = false	only set if the application didn specify a value for hibernate.session_factory_na

hibernate.ejb.entitymanager_factory_name = qualified persistence unit name	Is set to the application name persistence unit name (application can specify a different value but it needs to unique across all application deployments on the AS instance).
hibernate.query.jpaql_strict_compliance=true	
hibernate.auto_quote_keyword=false	
hibernate.implicit_naming_strategy =org.hibernate.boot.model.naming.ImplicitNamingStrategyJpaCompliantImpl	

In Hibernate 4.x (and greater), if **new_generator_mappings** is **true**:

- @GeneratedValue(AUTO) maps to org.hibernate.id.enhanced.SequenceStyleGenerator
- @GeneratedValue(TABLE) maps to org.hibernate.id.enhanced.TableGenerator
- @GeneratedValue(SEQUENCE) maps to org.hibernate.id.enhanced.SequenceStyleGenerator

In Hibernate 4.x (and greater), if **new_generator_mappings** is **false**:

- @GeneratedValue(AUTO) maps to Hibernate "native"
- @GeneratedValue(TABLE) maps to org.hibernate.id.MultipleHiLoPerTableGenerator
- @GeneratedValue(SEQUENCE) to Hibernate "seqhilo"

6.9.22 Persistence unit properties

The following properties are supported in the persistence unit definition (in the persistence.xml file):

Property	Purpose
jboss.as.jpa.providerModule	name of the persistence provider module (default is **org.hibernate**). Should be **application**, if a persistence provider is packaged with the application. See note below about some module names that are built in (based on the **provider**).
jboss.as.jpa.adapterModule	name of the integration classes that help WildFly to work with the persistence provider.
jboss.as.jpa.adapterClass	class name of the integration adapter.
jboss.as.jpa.managed	set to **false** to disable container managed JPA access to the persistence unit. The default is **true**, which enables container managed JPA access to the persistence unit. This is typically set to **false** for Spring applications.

jboss.as.jpa.classtransformer	set to **false** to disable class transformers for the persistence unit. Set to **true**, to allow entity class enhancing/rewriting.
wildfly.jpa.default-unit	set to **true** to choose the default persistence unit in an application. This is useful if you inject a persistence context without specifying the unitName (@PersistenceContext EntityManager em) but have multiple persistence units specified in your persistence.xml.
wildfly.jpa.twophasebootstrap	persistence providers (like Hibernate ORM 4.3+ via EntityManagerFactoryBuilder), allow a two phase persistence unit bootstrap, which improves JPA integration with CDI. Setting the **wildfly.jpa.twophasebootstrap** hint to false, disables the two phase bootstrap (for the persistence unit that contains the hint).
wildfly.jpa.allowdefaultdatasourceuse	set to false to prevent persistence unit from using the default data source. Defaults to true. This is only important for persistence units that do not specify a datasource.
jboss.as.jpa.deferdetach	Controls whether transaction scoped persistence context used in non-JTA transaction thread, will detach loaded entities after each EntityManager invocation or when the persistence context is closed (e.g. business method ends). Defaults to false (entities are cleared after EntityManager invocation) and if set to true, the detach is deferred until the context is closed.
wildfly.jpa.hibernate.search.module	Controls which version of Hibernate Search to include on classpath. Only makes sense when using Hibernate as JPA implementation. The default is **auto**; other valid values are **none** or a full module identifier to use an alternative version.
jboss.as.jpa.scopedname	Specify the qualified (application scoped) persistence unit name to be used. By default, this is internally set to the application name + persistence unit name. The hibernate.cache.region_prefix will default to whatever you set jboss.as.jpa.scopedname to. Make sure you set the jboss.as.jpa.scopedname value to a value not already in use by other applications deployed on the same application server instance.

wildfly.jpa.allowjoinedunsync	If set to true, allows an SynchronizationType.UNSYNCHRONIZED persistence context that has been joined to the active JTA transaction, to be propagated into a SynchronizationType.SYNCHRONIZED persistence context. Otherwise, an IllegalStateException exception would of been thrown that complains that an unsychronized persistence context cannot be propagated into a synchronized persistence context. Defaults to false.
wildfly.jpa.skipmixedsynctypechecking	Set to true to disable the throwing of an IllegalStateException exception when propagating an SynchronizationType.UNSYNCHRONIZED persistence context into a SynchronizationType.SYNCHRONIZED persistence context. This is a workaround intended to allow applications that used to incorrectly not get IllegalStateException exception with extended persistence contexts, to avoid the IllegalStateException, so they don't have to change their application right away (for compatibility purposes). This hint may be deprecated in a future release. See WFLY-7108 for more details. Defaults to false.

6.9.23 Determine the persistence provider module

As mentioned above, if the **jboss.as.jpa.providerModule** property is not specified, the provider module name is determined by the **provider** name specified in the persistence.xml. The mapping is:

Provider Name	Module name
blank	**org.hibernate**
org.hibernate.ejb.HibernatePersistence	**org.hibernate**
org.hibernate.ogm.jpa.HibernateOgmPersistence	**org.hibernate.ogm**
oracle.toplink.essentials.PersistenceProvider	**oracle.toplink**
oracle.toplink.essentials.ejb.cmp3.EntityManagerFactoryProvider	**oracle.toplink**
org.eclipse.persistence.jpa.PersistenceProvider	**org.eclipse.persistence**
org.datanucleus.api.jpa.PersistenceProviderImpl	**org.datanucleus**
org.datanucleus.store.appengine.jpa.DatastorePersistenceProvider	**org.datanucleus:appengine**
org.apache.openjpa.persistence.PersistenceProviderImpl	**org.apache.openjpa**

6.9.24 Binding EntityManagerFactory/EntityManager to JNDI

By default WildFly does **not** bind the entity manager factory to JNDI. However, you can explicitly configure this in the persistence.xml of your application by setting the `jboss.entity.manager.factory.jndi.name` hint. The value of that property should be the JNDI name to which the entity manager factory should be bound.

You can also bind a container managed (transaction scoped) entity manager to JNDI as well, }}via hint jboss.entity.manager.jndi.name{}{{. As a reminder, a transaction scoped entity manager (persistence context), acts as a proxy that always gets an unique underlying entity manager (at the persistence provider level).

Here's an example:

persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
   xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
        http://java.sun.com/xml/ns/persistence
        http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
   <persistence-unit name="myPU">
      <!-- If you are running in a production environment, add a managed
         data source, the example data source is just for proofs of concept! -->
      <jta-data-source>java:jboss/datasources/ExampleDS</jta-data-source>
      <properties>
         <!-- Bind entity manager factory to JNDI at java:jboss/myEntityManagerFactory -->
         <property name="jboss.entity.manager.factory.jndi.name"
value="java:jboss/myEntityManagerFactory" />
         <property name="jboss.entity.manager.jndi.name" value="java:/myEntityManager"/>
       </properties>
   </persistence-unit>
</persistence>
```

```
@Stateful
public class ExampleSFSB {
  public void createSomeEntityWithTransactionScopedEM(String name) {
    Context context = new InitialContext();
    javax.persistence.EntityManager entityManager = (javax.persistence.EntityManager)
context.lookup("java:/myEntityManager");
    SomeEntity someEntity = new SomeEntity();
    someEntity.setName(name);    entityManager.persist(name);
  }
}
```

6.9.25 Community

Many thanks to the community, for reporting issues, solutions and code changes. A number of people have been answering Wildfly forum questions related to JPA usage. I would like to thank them for this, as well as those reporting issues. For those of you that haven't downloaded the AS source code and started hacking patches together. I would like to encourage you to start by reading Hacking on WildFly. You will find that it easy very easy to find your way around the WildFly/JPA/* source tree and make changes. Also, new for WildFly, is the JipiJapa project that contains additional integration code that makes EE JPA application deployments work better. The following list of contributors should grow over time, I hope to see more of you listed here.

People who have contributed to the WildFly JPA layer:

- Carlo de Wolf (lead of the EJB3 project)
- Steve Ebersole (lead of the Hibernate ORM project)
- Stuart Douglas (lead of the Seam Persistence project, WildFly project team member/committer)
- Jaikiran Pai (Active member of JBoss forums and JBoss EJB3 project team member)
- Strong Liu (leads the productization effort of Hibernate in the EAP product)
- Scott Marlow (lead of the WildFly container JPA sub-project)
- Antti Laisi **(OpenJPA integration changes)**
- Galder Zamarreño (Infinispan 2lc documentation)
- Sanne Grinovero (lead of the Hibernate Search project)
- Paul Ferraro (Infinispan 2lc integration)

6.10 OSGi developer guide

Couldn't find a page to include called: OSGi Developer Guide

6.11 JNDI reference guide

6.11.1 Overview

WildFly offers several mechanisms to retrieve components by name. Every WildFly instance has it's own local JNDI namespace (`java:`) which is unique per JVM. The layout of this namespace is primarily governed by the Java EE specification. Applications which share the same WildFly instance can use this namespace to intercommunicate. In addition to local JNDI, a variety of mechanisms exist to access remote components.

- Client JNDI - This is a mechanism by which remote components can be accessed using the JNDI APIs, but ***without network round-trips***. This approach is the most efficient, and ***removes a potential single point of failure***. For this reason, it is highly recommended to use Client JNDI over traditional remote JNDI access. However, to make this possible, it does require that all names follow a strict layout, so user customizations are not possible. Currently only access to remote EJBs is supported via the `ejb:` namespace. Future revisions will likely add a JMS client JNDI namespace.
- Traditional Remote JNDI - This is a more familiar approach to EE application developers, where the client performs a remote component name lookup against a server, and a proxy/stub to the component is serialized as part of the name lookup and returned to the client. The client then invokes a method on the proxy which results in another remote network call to the underlying service. In a nutshell, traditional remote JNDI involves two calls to invoke an EE component, whereas Client JNDI requires one. It does however allow for customized names, and for a centralised directory for multiple application servers. This centralized directory is, however, *a single point of failure*.
- EE Application Client / Server-To-Server Delegation - This approach is where local names are bound as an *alias* to a remote name using one of the above mechanisms. This is useful in that it allows applications to only ever reference standard portable Java EE names in both code and deployment descriptors. It also allows for the application to be unaware of network topology details/ This can even work with Java SE clients by using the little known EE Application Client feature. This feature allows you to run an extremely minimal AS server around your application, so that you can take advantage of certain core services such as naming and injection.

6.11.2 Local JNDI

The Java EE platform specification defines the following JNDI contexts:

- `java:comp` - The namespace is scoped to the current component (i.e. EJB)
- `java:module` - Scoped to the current module
- `java:app` - Scoped to the current application
- `java:global` - Scoped to the application server

In addition to the standard namespaces, WildFly also provides the following two global namespaces:

- java:jboss
- java:/

Only entries within the `java:jboss/exported` context are accessible over remote JNDI.

For web deployments `java:comp` is aliased to `java:module`, so EJB's deployed in a war do not have their own comp namespace.

Binding entries to JNDI

There are several methods that can be used to bind entries into JNDI in WildFly.

Using a deployment descriptor

For Java EE applications the recommended way is to use a deployment descriptor to create the binding. For example the following `web.xml` binds the string `"Hello World"` to `java:global/mystring` and the string `"Hello Module"` to `java:comp/env/hello` (any non absolute JNDI name is relative to `java:comp/env` context).

```
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
         version="3.1">
    <env-entry>
        <env-entry-name>java:global/mystring</env-entry-name>
        <env-entry-type>java.lang.String</env-entry-type>
        <env-entry-value>Hello World</env-entry-value>
    </env-entry>
    <env-entry>
        <env-entry-name>hello</env-entry-name>
        <env-entry-type>java.lang.String</env-entry-type>
        <env-entry-value>Hello Module</env-entry-value>
    </env-entry>
</web-app>
```

For more details, see the Java EE Platform Specification.

Programatically

Java EE Applications

Standard Java EE applications may use the standard JNDI API, included with Java SE, to bind entries in the global namespaces (the standard `java:comp`, `java:module` and `java:app` namespaces are read-only, as mandated by the Java EE Platform Specification).

```
InitialContext initialContext = new InitialContext();
  initialContext.bind("java:global/a", 100);
```

There is no need to unbind entries created programatically, since WildFly tracks which bindings belong to a deployment, and the bindings are automatically removed when the deployment is undeployed.

WildFly Modules and Extensions

With respect to code in WildFly Modules/Extensions, which is executed out of a Java EE application context, using the standard JNDI API may result in a UnsupportedOperationException if the target namespace uses a WritableServiceBasedNamingStore. To work around that, the bind() invocation needs to be wrapped using WildFly proprietary APIs:

```
InitialContext initialContext = new InitialContext();
  WritableServiceBasedNamingStore.pushOwner(serviceTarget);
  try {
    initialContext.bind("java:global/a", 100);
  } finally {
    WritableServiceBasedNamingStore.popOwner();
  }
```

The ServiceTarget removes the bind when uninstalled, thus using one out of the module/extension domain usage should be avoided, unless entries are removed using unbind().

Naming Subsystem Configuration

It is also possible to bind to one of the three global namespaces using configuration in the naming subsystem. This can be done by either editing the `standalone.xml/domain.xml` file directly, or through the management API.

Four different types of bindings are supported:

- Simple - A primitive or java.net.URL entry (default is `java.lang.String`).
- Object Factory - This allows to to specify the `javax.naming.spi.ObjectFactory` that is used to create the looked up value.
- External Context - An external context to federate, such as an LDAP Directory Service
- Lookup - The allows to create JNDI aliases, when this entry is looked up it will lookup the target and return the result.

An example standalone.xml might look like:

```
<subsystem xmlns="urn:jboss:domain:naming:2.0" >
  <bindings>
    <simple name="java:global/a" value="100" type="int" />
    <simple name="java:global/jbossDocs" value="https://docs.jboss.org" type="java.net.URL" />
    <object-factory name="java:global/b" module="com.acme" class="org.acme.MyObjectFactory" />
    <external-context name="java:global/federation/ldap/example”
class="javax.naming.directory.InitialDirContext" cache="true">
      <environment>
        <property name="java.naming.factory.initial" value=“com.sun.jndi.ldap.LdapCtxFactory” />
        <property name="java.naming.provider.url" value=“ldap://ldap.example.com:389” />
        <property name="java.naming.security.authentication" value=“simple” />
        <property name="java.naming.security.principal" value=“uid=admin,ou=system” />
        <property name="java.naming.security.credentials" value=“secret” />
      </environment>
    </external-context>
    <lookup name="java:global/c" lookup="java:global/b" />
 </bindings>
</subsystem>
```

The CLI may also be used to bind an entry. As an example:

```
/subsystem=naming/binding=java\:global\/mybinding:add(binding-type=simple, type=long,
value=1000)
```

WildFly's Administrator Guide includes a section describing in detail the Naming subsystem configuration.

Retrieving entries from JNDI

Resource Injection

For Java EE applications the recommended way to lookup a JNDI entry is to use `@Resource` injection:

```
@Resource(lookup = "java:global/mystring")
  private String myString;

  @Resource(name = "hello")
  private String hello;

  @Resource
  ManagedExecutorService executor;
```

Note that `@Resource` is more than a JNDI lookup, it also binds an entry in the component's JNDI environment. The new bind JNDI name is defined by `@Resource`'s `name` attribute, which value, if unspecified, is the Java type concatenated with `/` and the field's name, for instance `java.lang.String/myString`. More, similar to when using deployment descriptors to bind JNDI entries. unless the name is an absolute JNDI name, it is considered relative to `java:comp/env`. For instance, with respect to the field named `myString` above, the `@Resource`'s `lookup` attribute instructs WildFly to lookup the value in `java:global/mystring`, bind it in `java:comp/env/java.lang.String/myString`, and then inject such value into the field.

With respect to the field named `hello`, there is no `lookup` attribute value defined, so the responsibility to provide the entry's value is delegated to the deployment descriptor. Considering that the deployment descriptor was the `web.xml` previously shown, which defines an environment entry with same `hello` name, then WildFly inject the valued defined in the deployment descriptor into the field.

The `executor` field has no attributes specified, so the bind's name would default to `java:comp/env/javax.enterprise.concurrent.ManagedExecutorService/executor`, but there is no such entry in the deployment descriptor, and when that happens it's up to WildFly to provide a default value or null, depending on the field's Java type. In this particular case WildFly would inject the default instance of a managed executor service, the value in `java:comp/DefaultManagedExecutorService`, as mandated by the EE Concurrency Utilities 1.0 Specification (JSR 236).

Standard Java SE JNDI API

Java EE applications may use, without any additional configuration needed, the standard JNDI API to lookup an entry from JNDI:

```
String myString = (String) new InitialContext().lookup("java:global/mystring");
```

or simply

```
String myString = InitialContext.doLookup("java:global/mystring");
```

6.11.3 Remote JNDI

WildFly supports two different types of remote JNDI. The old jnp based JNDI implementation used in JBoss AS versions prior to 7.x is no longer supported.

remote:

The `remote:` protocol uses the WildFly remoting protocol to lookup items from the servers local JNDI. To use it, you must have the appropriate jars on the class path, if you are maven user can be done simply by adding the following to your `pom.xml`:

```
<dependency>
  <groupId>org.wildfly</groupId>
  <artifactId>wildfly-ejb-client-bom</artifactId>
  <version>8.0.0.Final</version>
  <type>pom</type>
  <scope>compile</scope>
</dependency>
```

If you are not using maven a shaded jar that contains all required classes
can be found in the `bin/client` directory of WildFly's distribution.

```
final Properties env = new Properties();
env.put(Context.INITIAL_CONTEXT_FACTORY,
org.jboss.naming.remote.client.InitialContextFactory.class.getName());
env.put(Context.PROVIDER_URL, "remote://localhost:4447");
remoteContext = new InitialContext(env);
```

ejb:

The ejb: namespace is provided by the jboss-ejb-client library. This protocol allows you to look up EJB's, using their application name, module name, ejb name and interface type.

This is a client side JNDI implementation. Instead of looking up an EJB on the server the lookup name contains enough information for the client side library to generate a proxy with the EJB information. When you invoke a method on this proxy it will use the current EJB client context to perform the invocation. If the current context does not have a connection to a server with the specified EJB deployed then an error will occur. Using this protocol it is possible to look up EJB's that do not actually exist, and no error will be thrown until the proxy is actually used. The exception to this is stateful session beans, which need to connect to a server when they are created in order to create the session bean instance on the server.

Some examples are:

```
ejb:myapp/myejbjar/MyEjbName!com.test.MyRemoteInterface
ejb:myapp/myejbjar/MyStatefulName!comp.test.MyStatefulRemoteInterface?stateful
```

The first example is a lookup of a singleton, stateless or EJB 2.x home interface. This lookup will not hit the server, instead a proxy will be generated for the remote interface specified in the name. The second example is for a stateful session bean, in this case the JNDI lookup will hit the server, in order to tell the server to create the SFSB session.

For more details on how the server connections are configured, please see EJB invocations from a remote client using JNDI.

6.12 Spring applications development and migration guide

This document details the main points that need to be considered by Spring developers that wish to develop new applications or to migrate existing applications to be run into WildFly 8.

6.12.1 Dependencies and Modularity

WildFly 8 has a modular class loading strategy, different from previous versions of JBoss AS, which enforces a better class loading isolation between deployments and the application server itself. A detailed description can be found in the documentation dedicated to class loading in WildFly 8.

This reduces significantly the risk of running into a class loading conflict and allows applications to package their own dependencies if they choose to do so. This makes it easier for Spring applications that package their own dependencies - such as logging frameworks or persistence providers to run on WildFly 8.

At the same time, this does not mean that duplications and conflicts cannot exist on the classpath. Some module dependencies are implicit, depending on the type of deployment as shown here.

6.12.2 Persistence usage guide

Depending on the strategy being used, Spring applications can be:

- native Hibernate applications;
- JPA-based applications;
- native JDBC applications;

6.12.3 Native Spring/Hibernate applications

Applications that use the Hibernate API directly with Spring (i.e. through either one of LocalSessionFactoryBean or AnnotationSessionFactoryBean) may use a version of Hibernate 3 packaged inside the application. Hibernate 4 (which is provided through the 'org.hibernate' module of WildFly 8) is not supported by Spring 3.0 and Spring 3.1 (and may be supported by Spring 3.2 as described in SPR-8096), so adding this module as a dependency is not a solution.

6.12.4 based applications

Spring applications using JPA may choose between:

- using a server-deployed persistence unit;
- using a Spring-managed persistence unit.

Using server-deployed persistence units

Applications that use a server-deployed persistence unit must observe the typical Java EE rules in what concerns dependency management, i.e. the javax.persistence classes and persistence provider (Hibernate) are contained in modules which are added automatically by the application when the persistence unit is deployed.

In order to use the server-deployed persistence units from within Spring, either the persistence context or the persistence unit need to be registered in JNDI via web.xml as follows:

```
<persistence-context-ref>
    <persistence-context-ref-name>persistence/petclinic-em</persistence-unit-ref-name>
    <persistence-unit-name>petclinic</persistence-unit-name>
</persistence-context-ref>
```

or, respectively:

```
<persistence-unit-ref>
    <persistence-unit-ref-name>persistence/petclinic-emf</persistence-unit-ref-name>
    <persistence-unit-name>petclinic</persistence-unit-name>
</persistence-unit-ref>
```

When doing so, the persistence context or persistence unit are available to be looked up in JNDI, as follows:

```
<jee:jndi-lookup id="entityManager" jndi-name="java:comp/env/persistence/petclinic-em"
            expected-type="javax.persistence.EntityManager"/>
```

or

```
<jee:jndi-lookup id="entityManagerFactory" jndi-name="java:comp/env/persistence/petclinic-emf"
            expected-type="javax.persistence.EntityManagerFactory"/>
```

JNDI binding

JNDI binding via persistence.xml properties is not supported in WildFly 8.

Using Spring-managed persistence units

Spring applications running in WildFly 8 may also create persistence units on their own, using the LocalContainerEntityManagerFactoryBean. This is what these applications need to consider:

Placement of the persistence unit definitions

When the application server encounters a deployment that has a file named META-INF/persistence.xml (or, for that matter, WEB-INF/classes/META-INF/persistence.xml), it will attempt to create a persistence unit based on what is provided in the file. In most cases, such definition files are not compliant with the Java EE requirements, mostly because required elements such as the datasource of the persistence unit are supposed to be provided by the Spring context definitions, which will fail the deployment of the persistence unit, and consequently of the entire deployment.

Spring applications can easily avoid this type of conflict, by using a feature of the LocalContainerEntityManagerFactoryBean which is designed for this purpose. Persistence unit definition files can exist in other locations than META-INF/persistence.xml and the location can be indicated through the persistenceXmlLocation property of the factory bean class.

Assuming that the persistence unit is in the META-INF/jpa-persistence.xml, the corresponding definition can be:

```
<bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
       <property name="persistenceXmlLocation"
value="classpath*:META-INF/jpa-persistence.xml"/>
       <!-- other definitions -->
</bean>
```

Managing dependencies

Since the LocalContainerEntityManagerFactoryBean and the corresponding HibernateJpaVendorAdapter are based on Hibernate 3, it is required to use that version with the application. Therefore, the Hibernate 3 jars must be included in the deployment. At the same time, due the presence of @PersistenceUnit or @PersistenceContext annotations on the application classes, the application server will automatically add the 'org.hibernate' module as a dependency.

This can be avoided by instructing the server to exclude the module from the deployment's list of dependencies. In order to do so, include a META-INF/jboss-deployment-structure.xml or, for web applications, WEB-INF/jboss-deployment-structure.xml with the following content:

```
<jboss-deployment-structure xmlns="urn:jboss:deployment-structure:1.0">
  <deployment>
    <exclusions>
       <module name="org.hibernate"/>
    </exclusions>
  </deployment>
</jboss-deployment-structure>
```

6.13 All WildFly documentation

Couldn't find a page to include called: All JBoss AS 7 documentation

6.14 Application Client Reference

As a Java EE6 compliant server, WildFly 8 contains an application client. An application client is essentially a cut down server instance, that allow you to use EE features such as injection in a client side program.

This article is not a tutorial on application client development, rather it covers the specifics of the WildFly application client. There are tutorials available elsewhere that cover application client basics, such as this one.

Note that the application client is different to the EJB client libraries, it is perfectly possible to write client application that do not use the application client, but instead use the jboss-ejb-client library directly.

6.14.1 Getting Started

To launch the application client use the `appclient.sh` or `appclient.bat` script in the bin directory. For example:

```
./appclient.sh --host=10.0.0.1 myear.ear#appClient.jar arg1
```

The `--host` argument tells the appclient the server to connect to. The next argument is the application client deployment to use, application clients can only run a single deployment, and this deployment must also be deployed on the full server instance that the client is connecting too.

Any arguments after the deployment to use are passed directly through to the application clients `main` function.

6.14.2 Connecting to more than one host

If you want to connect to more than one host or make use of the clustering functionality then you need to specify a jboss-ejb-client.properties file rather than a host:

```
./appclient.sh --ejb-client-properties=my-jboss-ejb-client.properties myear.ear#appClient.jar
arg1
```

6.14.3 Example

A simple example how to package an application client and use it with WildFly can be within the quickstart appclient which is located on Github .

6.15 CDI Reference

WildFly uses Weld, the CDI reference implementation as its CDI provider. To activate CDI for a deployment simply add a `beans.xml` file in any archive in the deployment.

This document is not intended to be a CDI tutorial, it only covers CDI usage that is specific to WildFly. For some general information on CDI see the below links:

CDI Specification
Weld Reference Guide
The AS7 Quickstarts

6.15.1 Using CDI Beans from outside the deployment

For WildFly 8 onwards, it is now possible have classes outside the deployment be picked up as CDI beans. In order for this to work you must add a dependency on the external deployment that your beans are coming from, and make sure the META-INF directory of this deployment is imported, so that your deployment has visibility to the `beans.xml` file (To import beans from outside the deployment they must be in an archive with a `beans.xml` file).

There are two ways to do this, either using the `MANIFEST.MF` or using `jboss-deployment-structure.xml`.

Using `MANIFEST.MF` you need to add a `Dependencies` entry, with meta-inf specified after the entry, e.g.

```
Dependencies: com.my-cdi-module meta-inf, com.my-other-cdi-module meta-inf
```

Using `jboss-deployment-structure.xml` you need to add a dependency entry with `meta-inf="import"`, e.g.

```
<jboss-deployment-structure xmlns="urn:jboss:deployment-structure:1.2">
    <deployment>
        <dependencies>
            <module name="deployment.d1.jar" meta-inf="import"/>
        </dependencies>
    </deployment>
</jboss-deployment-structure>
```

Note that this can be used to create beans from both modules in the `modules` directory, and from other deployments.

For more information on class loading and adding dependencies to your deployment please see the Class Loading Guide

6.15.2 Suppressing implicit bean archives

CDI 1.1 brings new options to packaging of CDI-enabled applications. In addition to well-known explicit bean archives (basically any archive containing the **beans.xml** file) the specification introduces **implicit bean archives**.

An implicit bean archive is any archive that contains one or more classes annotated with a bean defining annotation (scope annotation) or one or more session beans. As a result, the beans.xml file is no longer required for CDI to work in your application.

In an implicit bean archive **only those classes** that are either annotated with bean defining annotations or are session beans are recognized by CDI as beans (other classes cannot be injected).

This has a side-effect, though. Libraries exist that make use of scope annotation (bean defining annotations) for their own convenience but are not designed to run with CDI support. Guava would be an example of such library. If your application bundles such library it will be recognized as a CDI archive and may fail the deployment.

Fortunately, WildFly makes it possible to suppress implicit bean archives and only enable CDI in archives that bundle the beans.xml file. There are two ways to achieve this:

deployment configuration

You can either set this up for your deployment only by adding the following content to the **META-INF/jboss-all.xml** file of your application:

```
<jboss xmlns="urn:jboss:1.0">
    <weld xmlns="urn:jboss:weld:1.0" require-bean-descriptor="true"/>
</jboss>
```

Global configuration

Alternatively, you may configure this for all deployments in your WildFly instance by executing the following command:

```
/subsystem=weld:write-attribute(name=require-bean-descriptor,value=true)
```

6.15.3 Development mode

WildFly 10 introduces a special mode for application development which allows you to inspect and monitor your CDI deployments. This mode is turned off by default and note that some features of the **development mode may have negative impact on the performance and/or functionality of the application**.

deployment configuration

You can enable it locally in your application `web.xml` by setting the Servlet initialization parameter `org.jboss.weld.development` to `true`:

```
<context-param>
   <param-name>org.jboss.weld.development</param-name>
   <param-value>true</param-value>
 </context-param>
```

Global configuration

Alternatively, you can enable it globally in Weld subsystem by setting `development-mode` attribute to `true`:

```
/subsystem=weld:write-attribute(name=development-mode,value=true)
```

For more details and example you can check Weld development mode.

Once the development mode is enabled you can check your applications CDI information using Weld Probe - Weld Probe.

6.15.4 portable mode

CDI 1.1 clarifies some aspects of how CDI protable extensions work. As a result, some extensions that do not use the API properly (but were tolerated in CDI 1.0 environment) may stop working with CDI 1.1.If this is the case of your application you will see an exception like this:

```
org.jboss.weld.exceptions.IllegalStateException: WELD-001332: BeanManager method getBeans() is
not available during application initialization
```

Fortunatelly, there is a non-portable mode available in WildFly which skips some of the API usage checks and therefore allows the legacy extensions to work as before.

Again, there are two ways to enable the non-portable mode:

deployment configuration

You can either set this up for your deployment only by adding the following content to the **META-INF/jboss-all.xml** file of your application:

```
<jboss xmlns="urn:jboss:1.0">
    <weld xmlns="urn:jboss:weld:1.0" non-portable-mode="true" />
</jboss>
```

Global configuration

Alternatively, you may configure this for all deployments in your WildFly instance by executing the following command:

```
/subsystem=weld:write-attribute(name=non-portable-mode,value=true)
```

Note that new portable extensions should always use the BeanManager API properly and thus never required the non-portable mode. The non-portable mode only exists to preserve compatibility with legacy extensions!

6.16 Class Loading in WildFly

Since JBoss AS 7, Class loading is considerably different to previous versions of JBoss AS. Class loading is based on the JBoss Modules project. Instead of the more familiar hierarchical class loading environment, WildFly's class loading is based on modules that have to define explicit dependencies on other modules. Deployments in WildFly are also modules, and do not have access to classes that are defined in jars in the application server unless an explicit dependency on those classes is defined.

6.16.1 Deployment Module Names

Module names for top level deployments follow the format `deployment.myarchive.war` while sub deployments are named like `deployment.myear.ear.mywar.war`.

This means that it is possible for a deployment to import classes from another deployment using the other deployments module name, the details of how to add an explicit module dependency are explained below.

6.16.2 Automatic Dependencies

Even though in WildFly modules are isolated by default, as part of the deployment process some dependencies on modules defined by the application server are set up for you automatically. For instance, if you are deploying a Java EE application a dependency on the Java EE API's will be added to your module automatically. Similarly if your module contains a beans.xml file a dependency on Weld will be added automatically, along with any supporting modules that weld needs to operate.

For a complete list of the automatic dependencies that are added, please see Implicit module dependencies for deployments.

Automatic dependencies can be excluded through the use of `jboss-deployment-structure.xml`.

6.16.3 Class Loading Precedence

A common source of errors in Java applications is including API classes in a deployment that are also provided by the container. This can result in multiple versions of the class being created and the deployment failing to deploy properly. To prevent this in WildFly, module dependencies are added in a specific order that should prevent this situation from occurring.

In order of highest priority to lowest priority

1. System Dependencies - These are dependencies that are added to the module automatically by the container, including the Java EE api's.
2. User Dependencies - These are dependencies that are added through `jboss-deployment-structure.xml` or through the `Dependencies:` manifest entry.
3. Local Resource - Class files packaged up inside the deployment itself, e.g. class files from `WEB-INF/classes` or `WEB-INF/lib` of a war.
4. Inter deployment dependencies - These are dependencies on other deployments in an ear deployment. This can include classes in an ear's lib directory, or classes defined in other ejb jars.

6.16.4 WAR Class Loading

The war is considered to be a single module, so classes defined in `WEB-INF/lib` are treated the same as classes in `WEB-INF/classes`. All classes packaged in the war will be loaded with the same class loader.

6.16.5 EAR Class Loading

Ear deployments are multi-module deployments. This means that not all classes inside an ear will necessarily have access to all other classes in the ear, unless explicit dependencies have been defined. By default the `EAR/lib` directory is a single module, and every WAR or EJB jar deployment is also a separate module. Sub deployments (wars and ejb-jars) always have a dependency on the parent module, which gives them access to classes in `EAR/lib`, however they do not always have an automatic dependency on each other. This behaviour is controlled via the `ear-subdeployments-isolated` setting in the ee subsystem configuration:

```
<subsystem xmlns="urn:jboss:domain:ee:1.0" >
  <ear-subdeployments-isolated>false</ear-subdeployments-isolated>
</subsystem>
```

By default this is set to false, which allows the sub-deployments to see classes belonging to other sub-deployments within the .ear.

For example, consider the following .ear deployment:

```
myapp.ear
 |
 |--- web.war
 |
 |--- ejb1.jar
 |
 |--- ejb2.jar
```

If the ear-subdeployments-isolated is set to false, then the classes in web.war can access classes belonging to ejb1.jar and ejb2.jar. Similarly, classes from ejb1.jar can access classes from ejb2.jar (and vice-versa).

The ear-subdeployments-isolated element value has no effect on the isolated classloader of the .war file(s). i.e. irrespective of whether this flag is set to true or false, the .war within a .ear will have a isolated classloader and other sub-deployments within that .ear will not be able to access classes from that .war. This is as per spec.

If the ear-subdeployments-isolated is set to true then no automatic module dependencies between the sub-deployments are set up. User must manually setup the dependency with `Class-Path` entries, or by setting up explicit module dependencies.

Portability

The Java EE specification says that portable applications should not rely on sub deployments having access to other sub deployments unless an explicit Class-Path entry is set in the MANIFEST.MF. So portable applications should always use Class-Path entry to explicitly state their dependencies.

It is also possible to override the ear-subdeployments-isolated element value at a per deployment level. See the section on jboss-deployment-structure.xml below.

`Dependencies:` Manifest Entries

Deployments (or more correctly modules within a deployment) may set up dependencies on other modules by adding a `Dependencies:` manifest entry. This entry consists of a comma separated list of module names that the deployment requires. The available modules can be seen under the `modules` directory in the application server distribution. For example to add a dependency on javassist and apache velocity you can add a manifest entry as follows:

```
Dependencies: org.javassist export,org.apache.velocity export services,org.antlr
```

Each dependency entry may also specify some of the following parameters by adding them after the module name:

- `export` This means that the dependencies will be exported, so any module that depends on this module will also get access to the dependency.
- `services` By default items in META-INF of a dependency are not accessible, this makes items from `META-INF/services` accessible so services in the modules can be loaded.
- `optional` If this is specified the deployment will not fail if the module is not available.
- `meta-inf` This will make the contents of the `META-INF` directory available (unlike `services`, which just makes `META-INF/services` available). In general this will not cause any deployment descriptors in META-INF to be processed, with the exception of `beans.xml`. If a `beans.xml` file is present this module will be scanned by Weld and any resulting beans will be available to the application.
- `annotations` If a jandex index has be created for the module these annotations will be merged into the deployments annotation index. The Jandex index can be generated using the Jandex ant task , and must be named `META-INF/jandex.idx`. Note that it is not necessary to break open the jar being indexed to add this to the modules class path, a better approach is to create a jar containing just this index, and adding it as an additional resource root in the `module.xml` file.

Adding a dependency to all modules in an EAR

Using the `export` parameter it is possible to add a dependency to all sub deployments in an ear. If a module is exported from a `Dependencies:` entry in the top level of the ear (or by a jar in the `ear/lib` directory) it will be available to all sub deployments as well.

To generate a MANIFEST.MF entry when using maven put the following in your pom.xml:

pom.xml

```
<build>
   ...
   <plugins>
     <plugin>
       <groupId>org.apache.maven.plugins</groupId>
       <artifactId>maven-war-plugin</artifactId>
       <configuration>
          <archive>
             <manifestEntries>
                <Dependencies>org.slf4j</Dependencies>
             </manifestEntries>
          </archive>
       </configuration>
     </plugin>
   </plugins>
</build>
```

If your deployment is a jar you must use the `maven-jar-plugin` rather than the `maven-war-plugin`.

Class Path Entries

It is also possible to add module dependencies on other modules inside the deployment using the `Class-Path` manifest entry. This can be used within an ear to set up dependencies between sub deployments, and also to allow modules access to additional jars deployed in an ear that are not sub deployments and are not in the `EAR/lib` directory. If a jar in the `EAR/lib` directory references a jar via `Class-Path:` then this additional jar is merged into the parent ear's module, and is accessible to all sub deployments in the ear.

6.16.6 Global Modules

It is also possible to set up global modules, that are accessible to all deployments. This is done by modifying the configuration file (standalone/domain.xml).

For example, to add javassist to all deployments you can use the following XML:

standalone.xml/domain.xml

```
<subsystem xmlns="urn:jboss:domain:ee:1.0" >
  <global-modules>
    <module name="org.javassist" slot="main" />
  </global-modules>
</subsystem>
```

Note that the `slot` field is optional and defaults to `main`.

6.16.7 JBoss Deployment Structure File

`jboss-deployment-structure.xml` is a JBoss specific deployment descriptor that can be used to control class loading in a fine grained manner. It should be placed in the top level deployment, in `META-INF` (or `WEB-INF` for web deployments). It can do the following:

- Prevent automatic dependencies from being added
- Add additional dependencies
- Define additional modules
- Change an EAR deployments isolated class loading behaviour
- Add additional resource roots to a module

An example of a complete `jboss-deployment-structure.xml` file for an ear deployment is as follows:

jboss-deployment-structure.xml

```
<jboss-deployment-structure>
  <!-- Make sub deployments isolated by default, so they cannot see each others classes without
a Class-Path entry -->
  <ear-subdeployments-isolated>true</ear-subdeployments-isolated>
  <!-- This corresponds to the top level deployment. For a war this is the war's module, for an
ear -->
  <!-- This is the top level ear module, which contains all the classes in the EAR's lib folder
-->
  <deployment>
     <!-- exclude-subsystem prevents a subsystems deployment unit processors running on a
deployment -->
     <!-- which gives basically the same effect as removing the subsystem, but it only affects
single deployment -->
     <exclude-subsystems>
        <subsystem name="resteasy" />
```

```
    </exclude-subsystems>
    <!-- Exclusions allow you to prevent the server from automatically adding some dependencies
-->
    <exclusions>
        <module name="org.javassist" />
    </exclusions>
    <!-- This allows you to define additional dependencies, it is the same as using the
Dependencies: manifest attribute -->
    <dependencies>
      <module name="deployment.javassist.proxy" />
      <module name="deployment.myjavassist" />
      <!-- Import META-INF/services for ServiceLoader impls as well -->
      <module name="myservicemodule" services="import"/>
    </dependencies>
    <!-- These add additional classes to the module. In this case it is the same as including
the jar in the EAR's lib directory -->
    <resources>
      <resource-root path="my-library.jar" />
    </resources>
  </deployment>
  <sub-deployment name="myapp.war">
    <!-- This corresponds to the module for a web deployment -->
    <!-- it can use all the same tags as the <deployment> entry above -->
    <dependencies>
      <!-- Adds a dependency on a ejb jar. This could also be done with a Class-Path entry -->
      <module name="deployment.myear.ear.myejbjar.jar" />
    </dependencies>
    <!-- Set's local resources to have the lowest priority -->
    <!-- If the same class is both in the sub deployment and in another sub deployment that -->
    <!-- is visible to the war, then the Class from the other deployment will be loaded,  -->
    <!-- rather than the class actually packaged in the war. -->
    <!-- This can be used to resolve ClassCastExceptions  if the same class is in multiple sub
deployments-->
    <local-last value="true" />
  </sub-deployment>
  <!-- Now we are going to define two additional modules -->
  <!-- This one is a different version of javassist that we have packaged -->
  <module name="deployment.myjavassist" >
    <resources>
     <resource-root path="javassist.jar" >
       <!-- We want to use the servers version of javassist.util.proxy.* so we filter it out-->
       <filter>
         <exclude path="javassist/util/proxy" />
       </filter>
     </resource-root>
    </resources>
  </module>
  <!-- This is a module that re-exports the containers version of javassist.util.proxy -->
  <!-- This means that there is only one version of the Proxy classes defined          -->
  <module name="deployment.javassist.proxy" >
    <dependencies>
      <module name="org.javassist" >
        <imports>
          <include path="javassist/util/proxy" />
          <exclude path="/**" />
        </imports>
      </module>
    </dependencies>
```

```
  </module>
</jboss-deployment-structure>
```

The xsd for jboss-deployment-structure.xml is available at https://github.com/wildfly/wildfly/blob/master/build/src/main/resources/docs/schema/jboss-deployment-

6.16.8 Accessing JDK classes

Not all JDK classes are exposed to a deployment by default. If your deployment uses JDK classes that are not exposed you can get access to them using jboss-deployment-structure.xml with system dependencies:

Using jboss-deployment-structure.xml to access JDK classes

```
<jboss-deployment-structure xmlns="urn:jboss:deployment-structure:1.1">
    <deployment>
        <dependencies>
            <system export="true">
                <paths>
                    <path name="com/sun/corba/se/spi/legacy/connection"/>
                </paths>
            </system>
        </dependencies>
    </deployment>
</jboss-deployment-structure>
```

6.16.9 The "jboss.api" property and application use of modules shipped with WildFly

The WildFly distribution includes a large number of modules, a great many of which are included for use by WildFly internals, with no testing of the appropriateness of their direct use by applications or any commitment to continue to ship those modules in future releases if they are no longer needed by the internals. So how can a user know whether it is advisable for their application to specify an explicit dependency on a module WildFly ships? The "jboss.api" property specified in the module's module.xml file can tell you:

Example declaration of the jboss.api property

```
<module xmlns="urn:jboss:module:1.3" name="com.google.guava">
    <properties>
        <property name="jboss.api" value="private"/>
    </properties>
```

If a module does not have a property element like the above, then it's equivalent to one with a value of "public".

Following are the meanings of the various values you may see for the jboss.api property:

Value	Meaning
public	May be explicitly depended upon by end user applications. Will continue to be available in future releases within the same major series and should not have incompatible API changes in future releases within the same minor series, and ideally not within the same major series.
private	Intended for internal use only. Only tested according to internal usage. May not be safe for end user applications to use directly. Could change significantly or be removed in a future release without notice.
unsupported	If you see this value in a module.xml in a WildFly release, please file a bug report, as it is not applicable in WildFly. In EAP it has a meaning equivalent to "private" but that does not mean the module is "private" in WildFly; it could very easily be "public".
preview	May be explicitly depended upon by end user applications, but there are no guarantees of continued availability in future releases or that there will not be incompatible API changes. This is not a common classification in WildFly. It is not used in WildFly 10.
deprecated	May be explicitly depended upon by end user applications. Stable and reliable but an alternative should be sought. Will be removed in a future major release.

Note that these definitions are only applicable to WildFly. In EAP and other Red Hat products based on WildFly the same classifiers are used, with generally similar meaning, but the precise meaning is per the definitions on the Red Hat customer support portal.

If an application declares a direct dependency on a module marked "private", "unsupported" or "deprecated", during deployment a WARN message will be logged. The logging will be in log categories "org.jboss.as.dependency.private", "org.jboss.as.dependency.unsupported" and "org.jboss.as.dependency.deprecated" respectively. These categories are not used for other purposes, so once you feel sufficiently warned the logging can be safely suppressed by turning the log level for the relevant category to ERROR or higher.

Other than the WARN messages noted above, declaring a direct dependency on a non-public module has no impact on how WildFly processes the deployment.

6.17 Deployment Descriptors used In WildFly

This page gives a list and a description of all the valid deployment descriptors that a WildFly deployment can use. This document is a work in progress.

Descriptor	Location	Specification	Description	Info
`jboss-deployment-structure.xml`	`META-INF` or `WEB-INF` of the top level deployment		This file can be used to control class loading for the deployment	Class Loadir WildFly

`beans.xml`	`WEB-INF` or `META-INF`	CDI	The presence of this descriptor (even if empty) activates CDI	Weld Refere Guide
`web.xml`	`WEB-INF`	Servlet	Web deployment descriptor	
`jboss-web.xml`	`WEB-INF`		JBoss Web deployment descriptor. This can be use to override settings from web.xml, and to set WildFly specific options	
`ejb-jar.xml`	`WEB-INF` of a war, or `META-INF` of an EJB jar	EJB	The EJB spec deployment descriptor	ejb-jar.xml s
`jboss-ejb3.xml`	`WEB-INF` of a war, or `META-INF` of an EJB jar		The JBoss EJB deployment descriptor, this can be used to override settings from `ejb-jar.xml`, and to set WildFly specific settings	
`application.xml`	`META-INF` of an EAR	Java EE Platform Specification		application.x schema

`jboss-app.xml`	`META-INF` of an EAR		JBoss application deployment descriptor, can be used to override settings application.xml, and to set WildFly specific settings	
`persistence.xml`	`META-INF`	JPA	JPA descriptor used for defining persistence units	Hibernate Re Guide
`jboss-ejb-client.xml`	`WEB-INF` of a war, or `META-INF` of an EJB jar		Remote EJB settings. This file is used to setup the EJB client context for a deployment that is used for remote EJB invocations	EJB invocati from a remot server instar
`jbosscmp-jdbc.xml`	`META-INF` of an EJB jar		CMP deployment descriptor. Used to map CMP entity beans to a database. The format is largely unchanged from previous versions.	

`ra.xml`	`META-INF` of a rar archive		Spec deployment descriptor for resource adaptor deployments	IronJacamar Reference G Schema
`ironjacamar.xml`	`META-INF` of a rar archive		JBoss deployment descriptor for resource adaptor deployments	IronJacamar Reference G
`*-jms.xml`	`META-INF` or `WEB-INF`		JMS message destination deployment descriptor, used to deploy message destinations with a deployment	
`*-ds.xml`	`META-INF` or `WEB-INF`		Datasource deployment descriptor, use to bundle datasources with a deployment	DataSource Configuratio
`application-client.xml`	`META-INF` of an application client jar	Java EE6 Platform Specification	The spec deployment descriptor for application client deployments	application-c schema
`jboss-client.xml`	`META-INF` of an application client jar		The WildFly specific deployment descriptor for application client deployments	

`jboss-webservices.xml`	`META-INF` for EJB webservice deployments or `WEB-INF` for POJO webservice deployments/EJB webservice endpoints bundled in .war		The JBossWS 4.0.x specific deployment descriptor for webservice endpoints	

6.18 Development Guidelines and Recommended Practices

The purpose of this page is to document tips and techniques that will assist developers in creating fast, secure, and reliable applications. It is also a place to note what you should **avoid** doing when developing applications.

6.19 EE Concurrency Utilities

6.19.1 Overview

EE Concurrency Utilities (JSR 236) is a technology introduced with Java EE 7, which adapts well known Java SE concurrency utilities to the Java EE application environment specifics. The Java EE application server is responsible for the creation (and shutdown) of every instance of the EE Concurrency Utilities, and provide these to the applications, ready to use.

The EE Concurrency Utilities support the propagation of the invocation context, capturing the existent context in the application threads to use in their own threads, the same way a logged-in user principal is propagated when a servlet invokes an EJB asynchronously. The propagation of the invocation context includes, by default, the class loading, JNDI and security contexts.

WildFly creates a single default instance of each EE Concurrency Utility type in all configurations within the distribution, as mandated by the specification, but additional instances, perhaps customised to better serve a specific usage, may be created through WildFly's EE Subsystem Configuration. To learn how to configure EE Concurrency Utilities please refer to EE Concurrency Utilities Configuration. Additionally, the EE subsystem configuration also includes the configuration of which instance should be considered the default instance mandated by the Java EE specification, and such configuration is covered by Default EE Bindings Configuration.

6.19.2 Context Service

The Context Service (`javax.enterprise.concurrent.ContextService`) is a brand new concurrency utility, which applications may use to build contextual proxies from existing objects.

A contextual proxy is an object that sets a invocation context, captured when created, whenever is invoked, before delegating the invocation to the original object.

Usage example:

```
public void onGet(...) {
  Runnable task = ...;
  Runnable contextualTask = contextService.createContextualProxy(task, Runnable.class);
  // ...
}
```

WildFly default configurations creates a single default instance of a Context Service, which may be retrieved through `@Resource` injection:

```
@Resource
private ContextService contextService;
```

To retrieve instead a non default Context Service instance, `@Resource`'s `lookup` attribute needs to specify the JNDI name used in the wanted instance configuration. WildFly will always inject the default instance, no matter what's the `name` attribute value, if the `lookup` attribute is not defined.

Applications may alternatively use instead the standard JNDI API:

```
ContextService contextService = InitialContext.doLookup("java:comp/DefaultContextService");
```

As mandated by the Java EE specification, the default Context Service instance's JNDI name is `java:comp/DefaultContextService`.

6.19.3 Managed Thread Factory

The Managed Thread Factory (`javax.enterprise.concurrent.ManagedThreadFactory`) allows Java EE applications to create Java threads. It is an extension of Java SE's Thread Factory (`java.util.concurrent.ThreadFactory`) adapted to the Java EE platform specifics.

Managed Thread Factory instances are managed by the application server, thus Java EE applications are forbidden to invoke any lifecycle related method.

In case the Managed Thread Factory is configured to use a Context Service, the application's thread context is captured when a thread creation is requested, and such context is propagated to the thread's Runnable execution.

Managed Thread Factory threads implement `javax.enterprise.concurrent.ManageableThread`, which allows an application to learn about termination status.

Usage example:

```
public void onGet(...) {
  Runnable task = ...;
  Thread thread = managedThreadFactory.newThread(task);
  thread.start();
    // ...
}
```

WildFly default configurations creates a single default instance of a Managed Thread Factory, which may be retrieved through `@Resource` injection:

```
@Resource
private ManagedThreadFactory managedThreadFactory;
```

To retrieve instead a non default Managed Thread Factory instance, `@Resource`'s `lookup` attribute needs to specify the JNDI name used in the wanted instance configuration. WildFly will always inject the default instance, no matter what's the `name` attribute value, in case the `lookup` attribute is not defined.

Applications may alternatively use instead the standard JNDI API:

```
ManagedThreadFactory managedThreadFactory =
InitialContext.doLookup("java:comp/DefaultManagedThreadFactory");
```

As mandated by the Java EE specification, the default Managed Thread Factory instance's JNDI name is `java:comp/DefaultManagedThreadFactory`.

6.19.4 Managed Executor Service

The Managed Executor Service (`javax.enterprise.concurrent.ManagedExecutorService`) allows Java EE applications to submit tasks for asynchronous execution. It is an extension of Java SE's Executor Service (`java.util.concurrent.ExecutorService`) adapted to the Java EE platform requirements.

Managed Executor Service instances are managed by the application server, thus Java EE applications are forbidden to invoke any lifecycle related method.

In case the Managed Executor Service is configured to use a Context Service, the application's thread context is captured when the task is submitted, and propagated to the executor thread responsible for the task execution.

Usage example:

```
public void onGet(...) {
    Runnable task = ...;
    Future future = managedExecutorService.submit(task);
    // ...
}
```

WildFly default configurations creates a single default instance of a Managed Executor Service, which may be retrieved through `@Resource` injection:

```
@Resource
private ManagedExecutorService managedExecutorService;
```

To retrieve instead a non default Managed Executor Service instance, `@Resource`'s `lookup` attribute needs to specify the JNDI name used in the wanted instance configuration. WildFly will always inject the default instance, no matter what's the `name` attribute value, in case the `lookup` attribute is not defined.

Applications may alternatively use instead the standard JNDI API:

```
ManagedExecutorService managedExecutorService =
InitialContext.doLookup("java:comp/DefaultManagedExecutorService");
```

As mandated by the Java EE specification, the default Managed Executor Service instance's JNDI name is `java:comp/DefaultManagedExecutorService`.

6.19.5 Managed Scheduled Executor Service

The Managed Scheduled Executor Service (`javax.enterprise.concurrent.ManagedScheduledExecutorService`) allows Java EE applications to schedule tasks for asynchronous execution. It is an extension of Java SE's Executor Service (`java.util.concurrent.ScheduledExecutorService`) adapted to the Java EE platform requirements.

Managed Scheduled Executor Service instances are managed by the application server, thus Java EE applications are forbidden to invoke any lifecycle related method.

In case the Managed Scheduled Executor Service is configured to use a Context Service, the application's thread context is captured when the task is scheduled, and propagated to the executor thread responsible for the task execution.

Usage example:

```
public void onGet(...) {
    Runnable task = ...;
    ScheduledFuture future = managedScheduledExecutorService.schedule(task, 60,
TimeUnit.SECONDS);
    // ...
}
```

WildFly default configurations creates a single default instance of a Managed Scheduled Executor Service, which may be retrieved through `@Resource` injection:

```
@Resource
private ManagedScheduledExecutorService managedScheduledExecutorService;
```

To retrieve instead a non default Managed Scheduled Executor Service instance, `@Resource`'s `lookup` attribute needs to specify the JNDI name used in the wanted instance configuration. WildFly will always inject the default instance, no matter what's the `name` attribute value, in case the `lookup` attribute is not defined.

Applications may alternatively use instead the standard JNDI API:

```
ManagedScheduledExecutorService managedScheduledExecutorService =
InitialContext.doLookup("java:comp/DefaultManagedScheduledExecutorService");
```

As mandated by the Java EE specification, the default Managed Scheduled Executor Service instance's JNDI name is `java:comp/DefaultManagedScheduledExecutorService`.

6.20 EJB 3 Reference Guide

This chapter details the extensions that are available when developing Enterprise Java Beans [tm] on WildFly 8.

Currently there is no support for configuring the extensions using an implementation specific descriptor file.

6.20.1 Resource Adapter for Message Driven Beans

Each Message Driven Bean must be connected to a resource adapter.

Specification of Resource Adapter using Metadata Annotations

The `ResourceAdapter` annotation is used to specify the resource adapter with which the MDB should connect.

The `value` of the annotation is the name of the deployment unit containing the resource adapter. For example `jms-ra.rar`.

For example:

```
@MessageDriven(messageListenerInterface = PostmanPat.class)
@ResourceAdapter("ejb3-rar.rar")
```

6.20.2 as Principal

Whenever a run-as role is specified for a given method invocation the default anonymous principal is used as the caller principal. This principal can be overridden by specifying a run-as principal.

Specification of Run-as Principal using Metadata Annotations

The `RunAsPrincipal` annotation is used to specify the run-as principal to use for a given method invocation.

The `value` of the annotation specifies the name of the principal to use. The actual type of the principal is undefined and should not be relied upon.

Using this annotation without specifying a run-as role is considered an error.

For example:

```
@RunAs("admin")
@RunAsPrincipal("MyBean")
```

6.20.3 Security Domain

Each Enterprise Java Bean [tm] can be associated with a security domain. Only when an EJB is associated with a security domain will authentication and authorization be enforced.

Specification of Security Domain using Metadata Annotations

The `SecurityDomain` annotation is used to specify the security domain to associate with the EJB.

The `value` of the annotation is the name of the security domain to be used.

For example:

```
@SecurityDomain("other")
```

6.20.4 Transaction Timeout

For any newly started transaction a transaction timeout can be specified in seconds.

When a transaction timeout of `0` is used, then the actual transaction timeout will default to the domain configured default.
TODO: add link to tx subsystem

Although this is only applicable when using transaction attribute `REQUIRED` or `REQUIRES_NEW` the application server will not detect invalid setups.

New Transactions

Take care that even when transaction attribute `REQUIRED` is specified, the timeout will only be applicable if a **new** transaction is started.

Specification of Transaction Timeout with Metadata Annotations

The `TransactionTimeout` annotation is used to specify the transaction timeout for a given method.

The `value` of the annotation is the timeout used in the given `unit` granularity. It must be a positive integer or 0. Whenever 0 is specified the default domain configured timeout is used.

The `unit` specifies the granularity of the `value`. The actual value used is converted to seconds. Specifying a granularity lower than `SECONDS` is considered an error, even when the computed value will result in an even amount of seconds.

For example:@TransactionTimeout(value = 10, unit = TimeUnit.SECONDS)

Specification of Transaction Timeout in the Deployment Descriptor

The `trans-timeout` element is used to define the transaction timeout for business, home, component, and message-listener interface methods; no-interface view methods; web service endpoint methods; and timeout callback methods.

The `trans-timeout` element resides in the `urn:trans-timeout` namespace and is part of the standard `container-transaction` element as defined in the jboss namespace.

For the rules when a `container-transaction` is applicable please refer to EJB 3.1 FR 13.3.7.2.1.

Example of trans-timeout

jboss-ejb3.xml

```
<jboss:ejb-jar xmlns:jboss="http://www.jboss.com/xml/ns/javaee"
               xmlns="http://java.sun.com/xml/ns/javaee"
               xmlns:tx="urn:trans-timeout"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee
http://www.jboss.org/j2ee/schema/jboss-ejb3-2_0.xsd
http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd
urn:trans-timeout http://www.jboss.org/j2ee/schema/trans-timeout-1_0.xsd"
               version="3.1"
               impl-version="2.0">
    <assembly-descriptor>
        <container-transaction>
            <method>
                <ejb-name>BeanWithTimeoutValue</ejb-name>
                <method-name>*</method-name>
                <method-intf>Local</method-intf>
            </method>
            <tx:trans-timeout>
                <tx:timeout>10</tx:timeout>
                <tx:unit>Seconds</tx:unit>
            </tx:trans-timeout>
        </container-transaction>
    </assembly-descriptor>
</jboss:ejb-jar>
```

6.20.5 Timer service

The service is responsible to call the registered timeout methods of the different session beans.

A persistent timer will be identified by the name of the EAR, the name of the sub-deployment JAR and the Bean's name.
If one of those names are changed (e.g. EAR name contain a version) the timer entry became orphaned and the timer event will not longer be fired.

Single event timer

The timer is will be started once at the specified time.

In case of a server restart the timeout method of a persistent timer will only be called directly if the specified time is elapsed.
If the timer is not persistent (since EJB3.1 see 18.2.3) it will be not longer available if JBoss is restarted or the application is redeployed.

Recurring timer

The timer will be started at the specified first occurrence and after that point at each time if the interval is elapsed.
If the timer will be started during the last execution is not finished the execution will be suppressed with a warning to avoid concurrent execution.

In case of server downtime for a persistent timer, the timeout method will be called only once if one, or more than one, interval is elapsed.
If the timer is not persistent (since EJB3.1 see 18.2.3) it will not longer be active after the server is restarted or the application is redeployed.

Calendar timer

The timer will be started if the schedule expression match. It will be automatically deactivated and removed if there will be no next expiration possible, i.e. If you set a specific year.

For example:

> *@Schedule(... dayOfMonth="1", month="1", year="2012")*
> *// start once at 01-01-2012 00:00:00*

Programmatic calendar timer

If the timer is persistent it will be fetched at server start and the missed timeouts are called concurrent.
If a persistent timer contains an end date it will be executed once nevertheless how many times the execution was missed. Also a retry will be suppressed if the timeout method throw an Exception.
In case of such expired timer access to the given Timer object might throw a NoMoreTimeoutExcption or NoSuchObjectException.

If the timer is non persistent it will not longer be active after the server is restarted or the application is redeployed.

TODO: clarify whether this should happen concurrently/blocked or even fired only once like a recurring timer!

Annotated calendar timer

If the timer is non persistent it will not activated for missed events during the server is down. In case of server start the timer is scheduled based on the @Schedule annotation.

If the timer is persistent (default if not deactivated by annotation) all missed events are fetched at server start and the annotated timeout method is called concurrent.

TODO: clarify whether this should happen concurrently/blocked or even fired only once like a recurring timer!

6.20.6 Container interceptors

Overview

JBoss AS versions prior to WildFly8 allowed a JBoss specific way to plug-in user application specific interceptors on the server side so that those interceptors get invoked during an EJB invocation. Such interceptors differed from the typical (portable) spec provided Java EE interceptors. The Java EE interceptors are expected to run after the container has done necessary invocation processing which involves security context propagation, transaction management and other such duties. As a result, these Java EE interceptors come too late into the picture, if the user applications have to intercept the call before certain container specific interceptor(s) are run.

Typical EJB invocation call path on the server

A typical EJB invocation looks like this:

Client application

```
MyBeanInterface bean = lookupBean();

bean.doSomething();
```

The invocation on the bean.doSomething() triggers the following (only relevant portion of the flow shown below):

1. WildFly specific interceptor (a.k.a container interceptor) 1
2. WildFly specific interceptor (a.k.a container interceptor) 2
3. ....
4. WildFly specific interceptor (a.k.a container interceptor) N
5. User application specific Java EE interceptor(s) (if any)
6. Invocation on the EJB instance's method

The WildFly specific interceptors include the security context propagation, transaction management and other container provided services. In some cases, the "`container interceptors`" (let's call them that) might even decide break the invocation flow and not let the invocation proceed (for example: due to the invoking caller not being among the allowed user roles who can invoke the method on the bean).

Previous versions of JBoss AS allowed a way to plug-in the user application specific interceptors (which relied on JBoss AS specific libraries) into this invocation flow so that they do run some application specific logic before the control reaches step#5 above. For example, AS5 allowed the use of JBoss AOP interceptors to do this.

WildFly 8 doesn't have such a feature.

Feature request for WildFly

There were many community users who requested for this feature to be made available in WildFly. As a result, https://issues.jboss.org/browse/AS7-5897 JIRA was raised. This feature is now implemented.

Configuring container interceptors

As you can see from the JIRA https://issues.jboss.org/browse/AS7-5897, one of the goals of this feature implementation was to make sure that we don't introduce any new WildFly specific library dependencies for the container interceptors. So we decided to allow the Java EE interceptors (which are just POJO classes with lifecycle callback annotations) to be used as container interceptors. As such you won't need any dependency on any WildFly specific libraries. That will allow us to support this feature for a longer time in future versions of WildFly.

Furthermore, configuring these container interceptors is similar to configuring the Java EE interceptors for EJBs. In fact, it uses the same xsd elements that are allowed in ejb-jar.xml for 3.1 version of ejb-jar deployment descriptor.

Container interceptors can only be configured via deployment descriptors. There's no annotation based way to configure container interceptors. This was an intentional decision, taken to avoid introducing any WildFly specific library dependency for the annotation.

Configuring the container interceptors can be done in jboss-ejb3.xml file, which then gets placed under the META-INF folder of the EJB deployment, just like the ejb-jar.xml. Here's an example of how the container interceptor(s) can be configured in jboss-ejb3.xml:

jboss-ejb3.xml

```
<jboss xmlns="http://www.jboss.com/xml/ns/javaee"
       xmlns:jee="http://java.sun.com/xml/ns/javaee"
       xmlns:ci ="urn:container-interceptors:1.0">

    <jee:assembly-descriptor>
        <ci:container-interceptors>
            <!-- Default interceptor -->
            <jee:interceptor-binding>
                <ejb-name>*</ejb-name>

<interceptor-class>org.jboss.as.test.integration.ejb.container.interceptor.ContainerInterceptorOne
</jee:interceptor-binding>
            <!-- Class level container-interceptor -->
            <jee:interceptor-binding>
                <ejb-name>AnotherFlowTrackingBean</ejb-name>

<interceptor-class>org.jboss.as.test.integration.ejb.container.interceptor.ClassLevelContainerInte
</jee:interceptor-binding>
            <!-- Method specific container-interceptor -->
            <jee:interceptor-binding>
                <ejb-name>AnotherFlowTrackingBean</ejb-name>

<interceptor-class>org.jboss.as.test.integration.ejb.container.interceptor.MethodSpecificContainer
<method>
                    <method-name>echoWithMethodSpecificContainerInterceptor</method-name>
                </method>
            </jee:interceptor-binding>
            <!-- container interceptors in a specific order -->
            <jee:interceptor-binding>
                <ejb-name>AnotherFlowTrackingBean</ejb-name>
                <interceptor-order>

<interceptor-class>org.jboss.as.test.integration.ejb.container.interceptor.ClassLevelContainerInte
<interceptor-class>org.jboss.as.test.integration.ejb.container.interceptor.MethodSpecificContainer
<interceptor-class>org.jboss.as.test.integration.ejb.container.interceptor.ContainerInterceptorOne
</interceptor-order>
                <method>
                    <method-name>echoInSpecificOrderOfContainerInterceptors</method-name>
                </method>
            </jee:interceptor-binding>
        </ci:container-interceptors>
    </jee:assembly-descriptor>
</jboss>
```

- The usage of urn:container-interceptors:1.0 namespace which allows the container-interceptors elements to be configured
- The container-interceptors element which contain the interceptor bindings
- The interceptor bindings themselves are the same elements as what the EJB3.1 xsd allows for standard Java EE interceptors
- The interceptors can be bound either to all EJBs in the deployment (using the the * wildcard) or individual bean level (using the specific EJB name) or at specific method level for the EJBs.

The xsd for the urn:container-interceptors:1.0 namespace is available here https://github.com/jbossas/jboss-as/blob/master/ejb3/src/main/resources/jboss-ejb-container-intercept

The interceptor classes themselves are simple POJOs and use the `@javax.annotation.AroundInvoke` to mark the around invoke method which will get invoked during the invocation on the bean. Here's an example of the interceptor:

Example of container interceptor

```
public class ClassLevelContainerInterceptor {
    @AroundInvoke
    private Object iAmAround(final InvocationContext invocationContext) throws Exception {
        return this.getClass().getName() + " " + invocationContext.proceed();
    }

}
```

Container interceptor positioning in the interceptor chain

The container interceptors configured for a EJB are guaranteed to be run before the WildFly provided security interceptors, transaction management interceptors and other such interceptors thus allowing the user application specific container interceptors to setup any relevant context data before the invocation proceeds.

Semantic difference between container interceptor(s) and Java EE interceptor(s) API

Although the container interceptors are modeled to be similar to the Java EE interceptors, there are some differences in the API semantics. One such difference is that invoking on javax.interceptor.InvocationContext.getTarget() method is illegal for container interceptors since these interceptors are invoked way before the EJB components are setup or instantiated.

Testcase

This testcase in the WildFly codebase can be used for reference for implementing container interceptors in user applications
https://github.com/jbossas/jboss-as/blob/master/testsuite/integration/basic/src/test/java/org/jboss/as/test/integ

6.20.7 EJB3 Clustered Database Timers

Overview

Wildfly now supports clustered database backed timers. The clustering support is provided through the database, and as a result it is not intended to be a super high performance solution that supports thousands of timers going off a second, however properly tuned it should provide sufficient performance for most use cases.

Note that database timers can also be used in non-clustered mode.

Note that for this to work correctly the underlying database must support the READ_COMMITTED or SERIALIZABLE isolation mode and the datasource must be configured accordingly

Setup

In order to use clustered timers it is necessary to add a database backed timer store. This can be done from the CLI with the following command:

```
/subsystem=ejb3/service=timer-service/database-data-store=my-clustered-store:add(allow-execution=t
datasource-jndi-name='java:/MyDatasource', refresh-interval=60000, database='postgresql',
partition='mypartition')
```

An explanation of the parameters is below:

- **allow-execution** - If this node is allowed to execute timers. If this is false then timers added on this node will be added to the database for another node to execute. This allows you to limit timer execution to a few nodes in a cluster, which can greatly reduce database load for large clusters.
- **datasource-jndi-name** - The datasource to use
- **refresh-interval** - The refresh interval in milliseconds. This is the period of time that must elapse before this node will check the database for new timers added by other nodes. A smaller value means that timers will be picked up more quickly, however it will result in more load on the database. This is most important to tune if you are adding timers that will expire quickly. If the node that added the timer cannot execute it (e.g. because it has failed or because allow-execution is false), this timer may not be executed until a node has refreshed.
- **database** - Define the type of database that is in use. Some SQL statements are customised by database, and this tells the data store which version of the SQL to use.
  Without this attribute the server try to detected the type automatically, current supported types are *postgresql, mysql, oracle, db2, hsql* and *h2*.
  Note that this SQL resides in the file *modules/system/layers/base/org/jboss/as/ejb3/main/timers/timer-sql.properties*
  And as such is it possible to modify the SQL that is executed or add support for new databases by adding new DB specific SQL to this file (if you do add support for a new database it would be greatly appreciated if you could contribute the SQL back to the project).

- **partition** - A node will only see timers from other nodes that have the same partition name. This allows you to break a large cluster up into several smaller clusters, which should improve performance. e.g. instead of having a cluster of 100 nodes, where all hundred are trying to execute and refresh the same timers, you can create 20 clusters of 5 nodes by giving ever group of 5 a different partition name.

Non clustered timers

Note that you can still use the database data store for non-clustered timers, in which case set the refresh interval to zero and make sure that every node has a unique partition name (or uses a different database).

Using clustered timers in a deployment

It is possible to use the data store as default for all applications by changing the default-data-store within the ejb3 subsystem:

```
<timer-service thread-pool-name="timer" default-data-store="clustered-store">
        <data-stores>
            <database-data-store name="clustered-store"
datasource-jndi-name="java:jboss/datasources/ExampleDS" partition="timer"/>
        </data-stores>
    </timer-service>
```

Another option is to use a separate data store for specific applications, all that is required is to set the timer data store name in jboss-ejb3.xml:

```
<?xml version="1.1" encoding="UTF-8"?>
<jboss:ejb-jar xmlns:jboss="http://www.jboss.com/xml/ns/javaee"
               xmlns="http://java.sun.com/xml/ns/javaee"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns:timer="urn:timer-service:1.0"
               xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee
http://www.jboss.org/j2ee/schema/jboss-ejb3-2_0.xsd
                     http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd"
               version="3.1"
               impl-version="2.0">
    <assembly-descriptor>
            <timer:timer>
                <ejb-name>*</ejb-name>
                <timer:persistence-store-name>my-clustered-store</timer:persistence-store-name>
            </timer:timer>
        </assembly-descriptor>
</jboss:ejb-jar>
```

Technical details

Internally every node that is allowed to execute timers schedules a timeout for every timer is knows about. When this timeout expires then this node attempts to 'lock' the timer, by updating its state to running. The query this executes looks like:

```
UPDATE JBOSS_EJB_TIMER SET TIMER_STATE=? WHERE ID=? AND TIMER_STATE<>? AND NEXT_DATE=?;
```

Due to the use of a transaction and READ_COMMITTED or SERIALIZABLE isolation mode only one node will succeed in updating the row, and this is the node that the timer will run on.

6.20.8 EJB3 subsystem configuration guide

This page lists the options that are available for configuring the EJB subsystem.

A complete example of the config is shown below, with a full explanation of each

```
<subsystem xmlns="urn:jboss:domain:ejb3:1.2">
  <session-bean>
    <stateless>
      <bean-instance-pool-ref pool-name="slsb-strict-max-pool"/>
    </stateless>
    <stateful default-access-timeout="5000" cache-ref="simple" clustered-cache-ref="clustered"/>
    <singleton default-access-timeout="5000"/>
  </session-bean>
  <mdb>
    <resource-adapter-ref resource-adapter-name="hornetq-ra"/>
    <bean-instance-pool-ref pool-name="mdb-strict-max-pool"/>
  </mdb>
  <entity-bean>
    <bean-instance-pool-ref pool-name="entity-strict-max-pool"/>
  </entity-bean>
  <pools>
    <bean-instance-pools>
      <strict-max-pool name="slsb-strict-max-pool" max-pool-size="20"
instance-acquisition-timeout="5" instance-acquisition-timeout-unit="MINUTES"/>
      <strict-max-pool name="mdb-strict-max-pool" max-pool-size="20"
instance-acquisition-timeout="5" instance-acquisition-timeout-unit="MINUTES"/>
      <strict-max-pool name="entity-strict-max-pool" max-pool-size="100"
instance-acquisition-timeout="5" instance-acquisition-timeout-unit="MINUTES"/>
  </bean-instance-pools>
  </pools>
  <caches>
    <cache name="simple" aliases="NoPassivationCache"/>
    <cache name="passivating" passivation-store-ref="file" aliases="SimpleStatefulCache"/>
    <cache name="clustered" passivation-store-ref="infinispan" aliases="StatefulTreeCache"/>
  </caches>
  <passivation-stores>
    <file-passivation-store name="file"/>
    <cluster-passivation-store name="infinispan" cache-container="ejb"/>
  </passivation-stores>
  <async thread-pool-name="default"/>
  <timer-service thread-pool-name="default">
    <data-store path="timer-service-data" relative-to="jboss.server.data.dir"/>
  </timer-service>
  <remote connector-ref="remoting-connector" thread-pool-name="default"/>
  <thread-pools>
    <thread-pool name="default">
      <max-threads count="10"/>
      <keepalive-time time="100" unit="milliseconds"/>
      </thread-pool>
  </thread-pools>
  <iiop enable-by-default="false" use-qualified-name="false"/>
  <in-vm-remote-interface-invocation pass-by-value="false"/> <!-- Warning see notes below about
possible issues -->
</subsystem>
```

<session-bean>

<stateless>

This element is used to configure the instance pool that is used by default for stateless session beans. If it is not present stateless session beans are not pooled, but are instead created on demand for every invocation. The instance pool can be overridden on a per deployment or per bean level using `jboss-ejb3.xml` or the `org.jboss.ejb3.annotation.Pool` annotation. The instance pools themselves are configured in the `<pools>` element.

<stateful>

This element is used to configure Stateful Session Beans.

- `default-access-timeout` This attribute specifies the default time concurrent invocations on the same bean instance will wait to acquire the instance lock. It can be overridden via the deployment descriptor or via the `javax.ejb.AccessTimeout` annotation.
- `cache-ref` This attribute is used to set the default cache for non-clustered beans. It can be overridden by `jboss-ejb3.xml`, or via the `org.jboss.ejb3.annotation.Cache` annotation.
- `clustered-cache-ref` This attribute is used to set the default cache for clustered beans.

<singleton>

This element is used to configure Singleton Session Beans.

- `default-access-timeout` This attribute specifies the default time concurrent invocations will wait to acquire the instance lock. It can be overridden via the deployment descriptor or via the `javax.ejb.AccessTimeout` annotation.

<mdb>

<resource-adaptor-ref>

This element sets the default resource adaptor for Message Driven Beans.

<bean-instance-pool-ref>

This element is used to configure the instance pool that is used by default for Message Driven Beans. If it is not present they are not pooled, but are instead created on demand for every invocation. The instance pool can be overridden on a per deployment or per bean level using `jboss-ejb3.xml` or the `org.jboss.ejb3.annotation.Pool` annotation. The instance pools themselves are configured in the `<pools>` element.

<entity-bean>

This element is used to configure the behavior for EJB2 EntityBeans.

<bean-instance-pool-ref>

This element is used to configure the instance pool that is used by default for Entity Beans. If it is not present they are not pooled, but are instead created on demand for every invocation. The instance pool can be overridden on a per deployment or per bean level using `jboss-ejb3.xml` or the `org.jboss.ejb3.annotation.Pool` annotation. The instance pools themselves are configured in the `<pools>` element.

<pools>

<caches>

<passivation-stores>

<async>

This element enables async EJB invocations. It is also used to specify the thread pool that these invocations will use.

<timer-service>

This element enables the EJB timer service. It is also used to specify the thread pool that these invocations will use.

<data-store>

This is used to configure the directory that persistent timer information is saved to.

<remote>

This is used to enable remote EJB invocations. It specifies the remoting connector to use (as defined in the remoting subsystem configuration), and the thread pool to use for remote invocations.

<thread-pools>

This is used to configure the thread pools used by async, timer and remote invocations.

<iiop>

This is used to enable IIOP (i.e. CORBA) invocation of EJB's. If this element is present then the JacORB subsystem must also be installed. It supports the following two attributes:

- `enable-by-default` If this is true then all EJB's with EJB2.x home interfaces are exposed via IIOP, otherwise they must be explicitly enabled via `jboss-ejb3.xml`.
- `use-qualified-name` If this is true then EJB's are bound to the corba naming context with a binding name that contains the application and modules name of the deployment (e.g. myear/myejbjar/MyBean), if this is false the default binding name is simply the bean name.

<in-vm-remote-interface-invocation>

By default remote interface invocations use pass by value, as required by the EJB spec. This element can use used to enable pass by reference, which can give you a performance boost. Note WildFly will do a shallow check to see if the caller and the EJB have access to the same class definitions, which means if you are passing something such as a List<MyObject>, WildFly only checks the List to see if it is the same class definition on the call & EJB side. If the top level class definition is the same, JBoss will make the call using pass by reference, which means that if MyObject or any objects beneath it are loaded from different classloaders, you would get a ClassCastException. If the top level class definitions are loaded from different classloaders, JBoss will use pass by value. JBoss cannot do a deep check of all of the classes to ensure no ClassCastExceptions will occur because doing a deep check would eliminate any performance boost you would have received by using call by reference. It is recommended that you configure pass by reference only on callers that you are sure will use the same class definitions and not globally. This can be done via a configuration in the jboss-ejb-client.xml as shown below.

To configure a caller/client use pass by reference, you configure your top level deployment with a META-INF/jboss-ejb-client.xml containing:

```
<jboss-ejb-client xmlns="urn:jboss:ejb-client:1.0">
    <client-context>
        <ejb-receivers local-receiver-pass-by-value="false"/>
    </client-context>
</jboss-ejb-client>
```

6.20.9 EJB IIOP Guide

Enabling IIOP

To enable IIOP you must have the JacORB subsystem installed, and the `<iiop/>` element present in the ejb3 subsystem configuration. The `standalone-full.xml` configuration that comes with the distribution has both of these enabled.

The `<iiop/>` element takes two attributes that control the default behaviour of the server, for full details see EJB3 subsystem configuration guide.

Enabling JTS

To enable JTS simply add a `<jts/>` element to the transactions subsystem configuration.

It is also necessary to enable the JacORB transactions interceptor as shown below.

```
<subsystem xmlns="urn:jboss:domain:jacorb:1.1">
  <orb>
    <initializers transactions="on"/>
  </orb>
</subsystem>
```

Dynamic Stub's

Downloading stubs directly from the server is no longer supported. If you do not wish to pre-generate your stub classes JDK Dynamic stubs can be used instead. The enable JDK dynamic stubs simply set the `com.sun.CORBA.ORBUseDynamicStub` system property to `true`.

Configuring EJB IIOP settings via jboss-ejb3.xml

TODO

6.20.10 EJB over HTTP

Beginning with Wildfly 11 it is now possible to use HTTP as the transport (instead of remoting) for remote EJB and JNDI invocations.

Everything mentioned below is applicable for both JNDI and EJB functionality.

Server Configuration

In order to configure the server the http-invoker needs to be enabled on each virtual host you wish to use in the Undertow subsystem. This is enabled by default in standard configs, but if it has been removed it can be added via:

```
/subsystem=undertow/server=default-server/host=default-host/setting=http-invoker:add(http-authenti
path='/wildfly-services')
```

The Hhttp-invoker takes two parameters, a path (which defaults to /wildfly-services) and a http-authentication-factory which must be a reference to an Elytron http-authentication-factory.

Note that any deployment that wishes to use this must use Elytron security with the same security domain that corresponds to the HTTP authentication factory.

Performing Invocations

The mechanism for performing invocations is exactly the same as for the remoting based EJB client, the only difference is that instead of a 'remote+http' URI you use a 'http' URI (which must include the path that was configured in the invoker). For example if you are currently using 'remote+http://localhost:8080' as the target URI, you would change this to 'http://localhost:8080/wildfly-services'.

Implementation details

The wire protocol is detailed at
https://github.com/wildfly/wildfly-http-client/blob/master/docs/src/main/asciidoc/wire-spec-v1.asciidoc

6.20.11 jboss-ejb3.xml Reference

`jboss-ejb3.xml` is a custom deployment descriptor that can be placed in either ejb-jar or war archives. If it is placed in an ejb-jar then it must be placed in the `META-INF` folder, in a web archive it must be placed in the `WEB-INF` folder.

The contents of `jboss-ejb3.xml` are merged with the contents of `ejb-jar.xml`, with the `jboss-ejb3.xml` items taking precedence.

Example File

A simple example is shown below:

```
<?xml version="1.1" encoding="UTF-8"?>
<jboss:ejb-jar xmlns:jboss="http://www.jboss.com/xml/ns/javaee"
               xmlns="http://java.sun.com/xml/ns/javaee"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns:s="urn:security:1.1"
               xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee
http://www.jboss.org/j2ee/schema/jboss-ejb3-2_0.xsd http://java.sun.com/xml/ns/javaee
http://www.jboss.org/j2ee/schema/jboss-ejb3-spec-2_0.xsd"
               version="3.1"
               impl-version="2.0">
    <enterprise-beans>
        <message-driven>
            <ejb-name>ReplyingMDB</ejb-name>

<ejb-class>org.jboss.as.test.integration.ejb.mdb.messagedestination.ReplyingMDB</ejb-class>
            <activation-config>
                <activation-config-property>

<activation-config-property-name>destination</activation-config-property-name>
                    <activation-config-property-value>java:jboss/mdbtest/messageDestinationQueue
                    </activation-config-property-value>
                </activation-config-property>
            </activation-config>
        </message-driven>
    </enterprise-beans>
    <assembly-descriptor>
        <s:security>
            <ejb-name>DDMyDomainSFSB</ejb-name>
            <s:security-domain>myDomain</s:security-domain>
            <s:run-as-principal>myPrincipal</s:run-as-principal>
        </s:security>
    </assembly-descriptor>
</jboss:ejb-jar>
```

As you can see the format is largely similar to `ejb-jar.xml`, in fact they even use the same namespaces, however `jboss-ejb3.xml` adds some additional namespaces of its own to allow for configuring non-spec info. The format of the standard `http://java.sun.com/xml/ns/javaee` is well documented elsewhere, this document will cover the non-standard namespaces.

Note that the namespace "http://www.jboss.com/xml/ns/javaee" is bound to "jboss-ejb3-spec-2_0.xsd": this file redefines some elements of "ejb-jar_3_1.xml"

The root namespace http://www.jboss.com/xml/ns/javaee

Assembly descriptor namespaces

The following namespaces can all be used in the `<assembly-descriptor>` element. They can be used to apply their configuration to a single bean, or to all beans in the deployment by using `*` as the `ejb-name`.

The security namespace urn:security

This allows you to set the security domain and the run-as principal for an EJB.

```
<s:security>
  <ejb-name>*</ejb-name>
  <s:security-domain>myDomain</s:security-domain>
  <s:run-as-principal>myPrincipal</s:run-as-principal>
</s:security>
```

The resource adaptor namespace urn:resource-adapter-binding

This allows you to set the resource adaptor for an MDB.

```
<r:resource-adapter-binding>
  <ejb-name>*</ejb-name>
  <r:resource-adapter-name>myResourceAdaptor</r:resource-adapter-name>
</r:resource-adapter-binding>
```

The IIOP namespace urn:iiop

The IIOP namespace is where IIOP settings are configured. As there are quite a large number of options these are covered in the IIOP guide.

The pool namespace urn:ejb-pool:1.0

This allows you to select the pool that is used by the SLSB or MDB. Pools are defined in the server configuration (i.e. `standalone.xml` or `domain.xml`)

```
<p:pool>
  <ejb-name>*</ejb-name>
  <p:bean-instance-pool-ref>my-pool</p:bean-instance-pool-ref>
</p:pool>
```

The cache namespace urn:ejb-cache:1.0

This allows you to select the cache that is used by the SFSB. Caches are defined in the server configuration (i.e. `standalone.xml` or `domain.xml`)

```
<c:cache>
  <ejb-name>*</ejb-name>
  <c:cache-ref>my-cache</c:cache-ref>
</c:cache>
```

The clustering namespace urn:clustering:1.0

This namespace is deprecated and as of WildFly 8 its use has no effect. The clustering behavior of EJBs is determined by the profile in use on the server.

6.20.12 Message Driven Beans Controlled Delivery

There are three mechanisms in Wildfly that allow controlling if a specific MDB is actively receiving or not messages:

- delivery active
- delivery groups
- clustered singleton

We will see each one of them in the following sections.

Delivery Active

Delivery active is simply an attribute associated with the MDB that indicates if the MDB is receiving messages or not. If an MDB is not currently receiving messages, the messages will be saved in the queue or topic for later, according to the rules of the topic/queue.

You can configure delivery active using xml or annotations, and you can change its value after deployment using the cli.

- jboss-ejb3.xml:

In the jboss-ejb3 xml file, configure the value of active as false to mark that the MDB will not be receiving messages as soon as it is deployed:

```
<?xml version="1.1" encoding="UTF-8"?>
<jboss:ejb-jar xmlns:jboss="http://www.jboss.com/xml/ns/javaee"
               xmlns="http://java.sun.com/xml/ns/javaee"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns:d="urn:delivery-active:1.1"
               xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee
http://www.jboss.org/j2ee/schema/jboss-ejb3-2_0.xsd http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd"               version="3.1"
               impl-version="2.0">
    <assembly-descriptor>
        <d:delivery>
            <ejb-name>HelloWorldQueueMDB</ejb-name>
            <d:active>false</d:active>
        </d:delivery>
    </assembly-descriptor>
</jboss:ejb-jar>
```

You can use a wildcard “*” in the place of ejb-name if you want to apply that active value to all MDBs in your application.

- annotation

Alternatively, you can use the org.jboss.ejb3.annotation.DeliveryActive annotation, as in the example below:

```
@MessageDriven(name = "HelloWorldMDB", activationConfig = {

 @ActivationConfigProperty(propertyName = "destinationType", propertyValue = "javax.jms.Queue"),

 @ActivationConfigProperty(propertyName = "destination", propertyValue =
"queue/HELLOWORLDMDBQueue"),

 @ActivationConfigProperty(propertyName = "acknowledgeMode", propertyValue = "Auto-acknowledge")
})

@DeliveryActive(false)

public class HelloWorldMDB implements MessageListener {
    public void onMessage(Message rcvMessage) {
      // ...
    }
}
```

Start-delivery and Stop-Delivery

These management operations dynamically change the value of the active attribute, enabling or disabling delivery for the MDB. at runtime To use them, connect to the Wildfly instance you want to manage, then enter the path of the MDB you want to manage delivery for:

```
[standalone@localhost:9990 /] cd
deployment=jboss-helloworld-mdb.war/subsystem=ejb3/message-driven-bean=HelloWorldMDB

[standalone@localhost:9990 message-driven-bean=HelloWorldMDB] :stop-delivery
{"outcome" => "success"}

[standalone@localhost:9990 message-driven-bean=HelloWorldMDB] :start-delivery
{"outcome" => "success"}
```

Delivery Groups

Delivery groups provide a straightforward way to manage delivery for a group of MDBs. Every MDB belonging to a delivery group has delivery active if and only if that group is active, and has delivery inactive whenever the group is not active.

You can add a delivery group to the ejb3 subsystem using either the subsystem xml or cli. Next, we will see examples of each case. In those examples we will add only a single delivery group, but keep in mind that you can add as many delivery groups as you need to a Wildfly instance.

- the ejb3 subsystem xml (located in your configuration xml, such as standalone.xml)

```
<subsystem xmlns="urn:jboss:domain:ejb3:4.0">
    ...
    <mdb>
        ...
        <delivery-groups>
            <delivery-group name="mdb-group-name" active="true"/>
        </delivery-groups>
    </mdb>
    ...
</subsystem>
```

The example above adds a delivery group named “mdb-group-name” (you can use whatever name suits you best as the group name). The “true” active attribute indicates that all MDBs belonging to that group will have delivery active right after deployment. If you mark that attribute as false, you are indicating that every MDB belonging to the group will not start receiving messages after deployment, a condition that will remain until the group becomes active.

- jboss-cli

You can add a mdb-delivery-group using the add command as below:

```
[standalone@localhost:9990 /] ./subsystem=ejb3/mdb-delivery-group=mdb-group-name:add
{"outcome" => "success"}
```

Reading and Writing the Delivery State of a Delivery Group

You can check whether delivery is active for a group by reading the active attribute, which defaults to true:

```
[standalone@localhost:9990 /]
./subsystem=ejb3/mdb-delivery-group=mdb-group-name:read-attribute(name=active)
{ "outcome" => "success", "result" => true }
```

To make the the delivery-group inactive, just write the active attribute with a false value:

```
[standalone@localhost:9990 /]
./subsystem=ejb3/mdb-delivery-group=mdb-group-name:write-attribute(name=active,value=false)
{"outcome" => "success"}

[standalone@localhost:9990 /]
./subsystem=ejb3/mdb-delivery-group=mdb-group-name:read-attribute(name=active)
{ "outcome" => "success", "result" => false }
```

To make it active again, write the attribute with a true value:

```
[standalone@localhost:9990 /]
./subsystem=ejb3/mdb-delivery-group=mdb-group-name:write-attribute(name=active,value=true)
{"outcome" => "success"}

[standalone@localhost:9990 /]
./subsystem=ejb3/mdb-delivery-group=mdb-group-name:read-attribute(name=active)
{ "outcome" => "success", "result" => true }
```

Using Delivery Groups

To mark that an MDB belongs to a delivery-group, declare so in the jboss-ejb3.xml file:

```
<?xml version="1.1" encoding="UTF-8"?>

<jboss:ejb-jar xmlns:jboss="http://www.jboss.com/xml/ns/javaee"
               xmlns="http://java.sun.com/xml/ns/javaee"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns:d="urn:delivery-active:1.1"
               xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee
http://www.jboss.org/j2ee/schema/jboss-ejb3-2_0.xsd http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd"
               version="3.1"
               impl-version="2.0">
    <assembly-descriptor>
        <d:delivery>
            <ejb-name>HelloWorldMDB</ejb-name>
            <d:group>mdb-delivery-group</d:group>
        </d:delivery>
    </assembly-descriptor>
</jboss:ejb-jar>
```

You can also use a wildcard to mark that all MDBs in your application belong to a delivery-group. In the following example, we add all MDBs in the application to group1, except for HelloWorldMDB, that is added to group2:

```
<?xml version="1.1" encoding="UTF-8"?>
<jboss:ejb-jar xmlns:jboss="http://www.jboss.com/xml/ns/javaee"
               xmlns="http://java.sun.com/xml/ns/javaee"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns:d="urn:delivery-active:1.1"
               xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee
http://www.jboss.org/j2ee/schema/jboss-ejb3-2_0.xsd http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd"
               version="3.1"
               impl-version="2.0">
    <assembly-descriptor>
        <d:delivery>
            <ejb-name>*</ejb-name>
            <d:group>group1</d:group>
        </d:delivery>
        <d:delivery>
            <ejb-name>HelloWorldMDB</ejb-name>
            <d:group>group2</d:group>
        </d:delivery>
    </assembly-descriptor>
</jboss:ejb-jar>
```

Another option is to use org.jboss.ejb3.annotation.DeliveryGroup annotation on each MDB class belonging to a group:

```
@MessageDriven(name = "HelloWorldQueueMDB", activationConfig = {
 @ActivationConfigProperty(propertyName = "destinationType", propertyValue = "javax.jms.Queue"),
 @ActivationConfigProperty(propertyName = "destination", propertyValue =
"queue/HELLOWORLDMDBQueue"),
 @ActivationConfigProperty(propertyName = "acknowledgeMode", propertyValue = "Auto-acknowledge")
})

@DeliveryGroup(“group2”)

public class HelloWorldMDB implements MessageListener {
    ...
}
```

A MDB cannot belong to more than one delivery group. Also, all the delivery-groups used by an application must be installed in the Wildfly server upon deployment, or the deployment will fail with a message stating that the delivery-group is missing.

Clustered Singleton Delivery

Delivery can be marked as singleton in a clustered environment. In this case, only one node in the cluster will have delivery active for that MDB, whereas in all other nodes, delivery will be inactive. This option can be used for applications that are deployed in all nodes of the cluster. Such applications will be active in all nodes of the cluster, except for the MDBs that are marked as clustered singleton. For those MDBs, only one cluster node will be processing their messages. In case that node stops, another node will have delivery activated, guaranteeing that there is always one node processing the messages. This node is what we call the MDB clustered singleton master node.

Notice that applications using clustered singleton delivery can only be deployed in clustered Wildfly servers (i.e., servers that are using the ha configuration).

To mark delivery as clustered singleton, you can use the jboss-ejb3.xml or the @ClusteredSingleton annotation:

- jboss-ejb3.xml:

```
<?xml version="1.1" encoding="UTF-8"?>
<jboss:ejb-jar xmlns:jboss="http://www.jboss.com/xml/ns/javaee"
               xmlns="http://java.sun.com/xml/ns/javaee"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns:c="urn:clustering:1.1"
               xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee
http://www.jboss.org/j2ee/schema/jboss-ejb3-2_0.xsd http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd"
               version="3.1"
               impl-version="2.0">
    <assembly-descriptor>
        <c:clustering>
            <ejb-name>HelloWorldMDB</ejb-name>
            <c:clustered-singleton>true</c:clustered-singleton>
        </c:clustering>
    </assembly-descriptor>
</jboss:ejb-jar>
```

As in the previous jboss-ejb3.xml examples, a wildcard can be used in the place of the ejb-name to indicate that all MDBs in the application are singleton clustered.

- annotation

You can use the org.jboss.ejb3.annotation.ClusteredSingleton annotation to mark an MDB as clustered singleton:

```
@MessageDriven(name = "HelloWorldQueueMDB", activationConfig = {
 @ActivationConfigProperty(propertyName = "destinationType", propertyValue = "javax.jms.Queue"),
 @ActivationConfigProperty(propertyName = "destination", propertyValue =
"queue/HELLOWORLDMDBQueue"),
 @ActivationConfigProperty(propertyName = "acknowledgeMode", propertyValue = "Auto-acknowledge")
})

@ClusteredSingleton

public class HelloWorldMDB implements MessageListener { ... }
```

Using Multiple MDB Delivery Control Mechanisms

The previous d eliver y control mechanisms can be used together in a single MDB. In this case, they work as a set of restrictions for delivery to be active in a MDB.

For example, if an MDB belongs to a delivery group and is also a clustered singleton MDB, the delivery will be active for that MDB only if the delivery group is active in the cluster node that was elected as the singleton master.

Also, if you use jboss-cli to stopDelivery on a MDB that belongs to a delivery group, the MDB will stop receiving messages in case that group was active. If that group was not active, the MDB will continue in the same, inactive state. But, once that group is active, the MDB will not receive messages, unless a startDelivery operation is executed to revert the previously exectued stopDelivery operation.

Invoking stopDelivery on an MDB that is marked as clustered singleton will work in a similar way: no visible effect if the current node is not the clustered singleton master; but it will stop delivery of messages for that MDB if the current node is the clustered singleton master. If the current node is not the master, but eventually becomes so, the delivery of messages will not be active for that MDB, unless a startDelivery operation is invoked.

In other words, when more than one delivery control mechanism is used in conjunction, they act as a set of restrictions that need all to be true in order for the MDB to receive messages:

- **delivery-group + stop-delivery**: the delivery group needs to be active and the delivery needs to be started in order for that MDB to start receiving messages;
- **delivery-group + clustered singleton**: the delivery group needs to be active and the current node needs to be the clustered singleton master node in order for that MDB to start receiving messages;
- **delivery-group + clustered singleton + stop-delivery**: as above, delivery-group active, current node equals the clustered singleton master node, plus, start-delivery needs to be invoked on that MDB, only with these three factors being true the MDB will start receiving messages.

6.20.13 Securing EJBs

Overview

The Java EE spec specifies certain annotations (like @RolesAllowed, @PermitAll, @DenyAll) which can be used on EJB implementation classes and/or the business method implementations of the beans. Like with all other configurations, these security related configurations can also be done via the deployment descriptor (ejb-jar.xml). We *won't* be going into the details of Java EE specific annotations/deployment descriptor configurations in this chapter but instead will be looking at the vendor specific extensions to the security configurations.

Security Domain

The Java EE spec doesn't mandate a specific way to configure security domain for a bean. It leaves it to the vendor implementations to allow such configurations, the way they wish. In WildFly 8, the use of `@org.jboss.ejb3.annotation.SecurityDomain` annotation allows the developer to configure the security domain for a bean. Here's an example:

```
import org.jboss.ejb3.annotation.SecurityDomain;

import javax.ejb.Stateless;

@Stateless

@SecurityDomain("other")

public class MyBean ...

{

....
```

The use of @SecurityDomain annotation lets the developer to point the container to the name of the security domain which is configured in the EJB3 subsystem in the standalone/domain configuration. The configuration of the security domain in the EJB3 subsystem is out of the scope of this chapter.

An alternate way of configuring a security domain, instead of using annotation, is to use jboss-ejb3.xml deployment descriptor. Here's an example of how the configuration will look like:

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss:jboss
        xmlns="http://java.sun.com/xml/ns/javaee"
        xmlns:jboss="http://www.jboss.com/xml/ns/javaee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:s="urn:security:1.1"
        xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee
http://www.jboss.org/j2ee/schema/jboss-ejb3-2_0.xsd http://java.sun.com/xml/ns/javaee
http://www.jboss.org/j2ee/schema/jboss-ejb3-spec-2_0.xsd"
        version="3.1" impl-version="2.0">

    <assembly-descriptor>
        <s:security>
            <!-- Even wildcard * is supported -->
            <ejb-name>MyBean</ejb-name>
            <!-- Name of the security domain which is configured in the EJB3 subsystem -->
            <s:security-domain>other</s:security-domain>
        </s:security>
    </assembly-descriptor>
</jboss:jboss>
```

As you can see we use the security-domain element to configure the security domain.

The jboss-ejb3.xml is expected to be placed in the .jar/META-INF folder of a .jar deployment or .war/WEB-INF folder of a .war deployment.

Absence of security domain configuration but presence of other security metadata

Let's consider the following example bean:

```
@Stateless
public class FooBean {

 @RolesAllowed("bar")
 public void doSomething() {
  ..
 }
...
}
```

As you can see the `doSomething` method is configured to be accessible for users with role "bar". However, the bean isn't configured for any specific security domain. Prior to WildFly 8, the absence of an explicitly configured security domain on the bean would leave the bean unsecured, which meant that even if the `doSomething` method was configured with `@RolesAllowed("bar")` anyone even without the "bar" role could invoke on the bean.

In WildFly 8, the presence of any security metadata (like @RolesAllowed, @PermitAll, @DenyAll, @RunAs, @RunAsPrincipal) on the bean or any business method of the bean, makes the bean secure, even in the absence of an explicitly configured security domain. In such cases, the security domain name is default to "other". Users can explicitly configure an security domain for the bean if they want to using either the annotation or deployment descriptor approach explained earlier.

Access to methods without explicit security metadata, on a secured bean

Consider this example bean:

```
@Stateless
public class FooBean {

 @RolesAllowed("bar")
 public void doSomething() {
  ..
 }


 public void helloWorld() {
  ...
    }
}
```

As you can see the `doSomething` method is marked for access for only users with role "bar". That enables security on the bean (with security domain defaulted to "other"). However, notice that the method `helloWorld` doesn't have any specific security configurations.

In WildFly 8, such methods which have no explicit security configurations, in a secured bean, will be treated similar to a method with `@DenyAll` configuration. What that means is, no one is allowed access to the `helloWorld` method. This behaviour can be controlled via the jboss-ejb3.xml deployment descriptor at a per bean level or a per deployment level as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss:jboss
        xmlns="http://java.sun.com/xml/ns/javaee"
        xmlns:jboss="http://www.jboss.com/xml/ns/javaee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:s="urn:security:1.1"
        xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee
http://www.jboss.org/j2ee/schema/jboss-ejb3-2_0.xsd http://java.sun.com/xml/ns/javaee
http://www.jboss.org/j2ee/schema/jboss-ejb3-spec-2_0.xsd"
        version="3.1" impl-version="2.0">

    <assembly-descriptor>
        <s:security>
            <!-- Even wildcard * is supported where * is equivalent to all EJBs in the
deployment -->
            <ejb-name>FooBean</ejb-name>

<s:missing-method-permissions-deny-access>false</s:missing-method-permissions-deny-access>
        </s:security>
    </assembly-descriptor>
</jboss:jboss>
```

Notice the use of `<missing-method-permissions-deny-access>` element. The value for this element can either be true or false. If this element isn't configured then it is equivalent to a value of true i.e. no one is allowed access to methods, which have no explicit security configurations, on secured beans. Setting this to false allows access to such methods for all users i.e. the behaviour will be switched to be similar to `@PermitAll`.

This behaviour can also be configured at the EJB3 subsystem level so that it applies to all EJB3 deployments on the server, as follows:

```
<subsystem xmlns="urn:jboss:domain:ejb3:1.4">
...
            <default-missing-method-permissions-deny-access value="true"/>
...
</subsystem>
```

Again, the `default-missing-method-permissions-deny-access` element accepts either a true or false value. A value of true makes the behaviour similar to `@DenyAll` and a value of false makes it behave like `@PermitAll`

6.21 EJB invocations from a remote client using JNDI

This chapter explains how to invoke EJBs from a remote client by using the JNDI API to first lookup the bean proxy and then invoke on that proxy.

After you have read this article, do remember to take a look at Remote EJB invocations via JNDI - EJB client API or remote-naming project

Before getting into the details, we would like the users to know that we have introduced a new EJB client API, which is a WildFly-specific API and allows invocation on remote EJBs. This client API isn't based on JNDI. So remote clients need not rely on JNDI API to invoke on EJBs. A separate document covering the EJB remote client API will be made available. For now, you can refer to the javadocs of the EJB client project at http://docs.jboss.org/ejbclient/. In this document, we'll just concentrate on the traditional JNDI based invocation on EJBs. So let's get started:

6.21.1 Deploying your EJBs on the server side:

Users who already have EJBs deployed on the server side can just skip to the next section.

As a first step, you'll have to deploy your application containing the EJBs on the Wildfly server. If you want those EJBs to be remotely invocable, then you'll have to expose at least one remote view for that bean. In this example, let's consider a simple Calculator stateless bean which exposes a RemoteCalculator remote business interface. We'll also have a simple stateful CounterBean which exposes a RemoteCounter remote business interface. Here's the code:

```
package org.jboss.as.quickstarts.ejb.remote.stateless;

/**
 * @author Jaikiran Pai
 */
public interface RemoteCalculator {

    int add(int a, int b);

    int subtract(int a, int b);
}
```

```
package org.jboss.as.quickstarts.ejb.remote.stateless;

import javax.ejb.Remote;
import javax.ejb.Stateless;

/**
 * @author Jaikiran Pai
 */
@Stateless
@Remote(RemoteCalculator.class)
public class CalculatorBean implements RemoteCalculator {

    @Override
    public int add(int a, int b) {
        return a + b;
    }

    @Override
    public int subtract(int a, int b) {
        return a - b;
    }
}
```

```
package org.jboss.as.quickstarts.ejb.remote.stateful;

/**
 * @author Jaikiran Pai
 */
public interface RemoteCounter {

    void increment();

    void decrement();

    int getCount();
}
```

```
package org.jboss.as.quickstarts.ejb.remote.stateful;

import javax.ejb.Remote;
import javax.ejb.Stateful;

/**
 * @author Jaikiran Pai
 */
@Stateful
@Remote(RemoteCounter.class)
public class CounterBean implements RemoteCounter {

    private int count = 0;

    @Override
    public void increment() {
        this.count++;
    }

    @Override
    public void decrement() {
        this.count--;
    }

    @Override
    public int getCount() {
        return this.count;
    }
}
```

Let's package this in a jar (how you package it in a jar is out of scope of this chapter) named "jboss-as-ejb-remote-app.jar" and deploy it to the server. Make sure that your deployment has been processed successfully and there aren't any errors.

6.21.2 Writing a remote client application for accessing and invoking the EJBs deployed on the server

The next step is to write an application which will invoke the EJBs that you deployed on the server. In WildFly, you can either choose to use the WildFly specific EJB client API to do the invocation or use JNDI to lookup a proxy for your bean and invoke on that returned proxy. In this chapter we will concentrate on the JNDI lookup and invocation and will leave the EJB client API for a separate chapter.

So let's take a look at what the client code looks like for looking up the JNDI proxy and invoking on it. Here's the entire client code which invokes on a stateless bean:

```
package org.jboss.as.quickstarts.ejb.remote.client;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
```

```
import java.security.Security;
import java.util.Hashtable;

import org.jboss.as.quickstarts.ejb.remote.stateful.CounterBean;
import org.jboss.as.quickstarts.ejb.remote.stateful.RemoteCounter;
import org.jboss.as.quickstarts.ejb.remote.stateless.CalculatorBean;
import org.jboss.as.quickstarts.ejb.remote.stateless.RemoteCalculator;
import org.jboss.sasl.JBossSaslProvider;

/**
 * A sample program which acts a remote client for a EJB deployed on Wildfly 10 server.
 * This program shows how to lookup stateful and stateless beans via JNDI and then invoke on
them
 *
 * @author Jaikiran Pai
 */
public class RemoteEJBClient {

    public static void main(String[] args) throws Exception {
        // Invoke a stateless bean
        invokeStatelessBean();

        // Invoke a stateful bean
        invokeStatefulBean();
    }

    /**
     * Looks up a stateless bean and invokes on it
     *
     * @throws NamingException
     */
    private static void invokeStatelessBean() throws NamingException {
        // Let's lookup the remote stateless calculator
        final RemoteCalculator statelessRemoteCalculator = lookupRemoteStatelessCalculator();
        System.out.println("Obtained a remote stateless calculator for invocation");
        // invoke on the remote calculator
        int a = 204;
        int b = 340;
        System.out.println("Adding " + a + " and " + b + " via the remote stateless calculator
deployed on the server");
        int sum = statelessRemoteCalculator.add(a, b);
        System.out.println("Remote calculator returned sum = " + sum);
        if (sum != a + b) {
            throw new RuntimeException("Remote stateless calculator returned an incorrect sum "
+ sum + " ,expected sum was " + (a + b));
        }
        // try one more invocation, this time for subtraction
        int num1 = 3434;
        int num2 = 2332;
        System.out.println("Subtracting " + num2 + " from " + num1 + " via the remote stateless
calculator deployed on the server");
        int difference = statelessRemoteCalculator.subtract(num1, num2);
        System.out.println("Remote calculator returned difference = " + difference);
        if (difference != num1 - num2) {
            throw new RuntimeException("Remote stateless calculator returned an incorrect
difference " + difference + " ,expected difference was " + (num1 - num2));
        }
    }
```

```
    /**
     * Looks up a stateful bean and invokes on it
     *
     * @throws NamingException
     */
    private static void invokeStatefulBean() throws NamingException {
        // Let's lookup the remote stateful counter
        final RemoteCounter statefulRemoteCounter = lookupRemoteStatefulCounter();
        System.out.println("Obtained a remote stateful counter for invocation");
        // invoke on the remote counter bean
        final int NUM_TIMES = 20;
        System.out.println("Counter will now be incremented " + NUM_TIMES + " times");
        for (int i = 0; i < NUM_TIMES; i++) {
            System.out.println("Incrementing counter");
            statefulRemoteCounter.increment();
            System.out.println("Count after increment is " + statefulRemoteCounter.getCount());
        }
        // now decrementing
        System.out.println("Counter will now be decremented " + NUM_TIMES + " times");
        for (int i = NUM_TIMES; i > 0; i--) {
            System.out.println("Decrementing counter");
            statefulRemoteCounter.decrement();
            System.out.println("Count after decrement is " + statefulRemoteCounter.getCount());
        }
    }

    /**
     * Looks up and returns the proxy to remote stateless calculator bean
     *
     * @return
     * @throws NamingException
     */
    private static RemoteCalculator lookupRemoteStatelessCalculator() throws NamingException {
        final Hashtable jndiProperties = new Hashtable();
        jndiProperties.put(Context.URL_PKG_PREFIXES, "org.jboss.ejb.client.naming");
        final Context context = new InitialContext(jndiProperties);
        // The app name is the application name of the deployed EJBs. This is typically the ear
name
        // without the .ear suffix. However, the application name could be overridden in the
application.xml of the
        // EJB deployment on the server.
        // Since we haven't deployed the application as a .ear, the app name for us will be an
empty string
        final String appName = "";
        // This is the module name of the deployed EJBs on the server. This is typically the jar
name of the
        // EJB deployment, without the .jar suffix, but can be overridden via the ejb-jar.xml
        // In this example, we have deployed the EJBs in a jboss-as-ejb-remote-app.jar, so the
module name is
        // jboss-as-ejb-remote-app
        final String moduleName = "jboss-as-ejb-remote-app";
        // AS7 allows each deployment to have an (optional) distinct name. We haven't specified
a distinct name for
        // our EJB deployment, so this is an empty string
        final String distinctName = "";
        // The EJB name which by default is the simple class name of the bean implementation
class
```

```
        final String beanName = CalculatorBean.class.getSimpleName();
        // the remote view fully qualified class name
        final String viewClassName = RemoteCalculator.class.getName();
        // let's do the lookup
        return (RemoteCalculator) context.lookup("ejb:" + appName + "/" + moduleName + "/" +
distinctName + "/" + beanName + "!" + viewClassName);
    }

    /**
     * Looks up and returns the proxy to remote stateful counter bean
     *
     * @return
     * @throws NamingException
     */
    private static RemoteCounter lookupRemoteStatefulCounter() throws NamingException {
        final Hashtable jndiProperties = new Hashtable();
        jndiProperties.put(Context.URL_PKG_PREFIXES, "org.jboss.ejb.client.naming");
        final Context context = new InitialContext(jndiProperties);
        // The app name is the application name of the deployed EJBs. This is typically the ear
name
        // without the .ear suffix. However, the application name could be overridden in the
application.xml of the
        // EJB deployment on the server.
        // Since we haven't deployed the application as a .ear, the app name for us will be an
empty string
        final String appName = "";
        // This is the module name of the deployed EJBs on the server. This is typically the jar
name of the
        // EJB deployment, without the .jar suffix, but can be overridden via the ejb-jar.xml
        // In this example, we have deployed the EJBs in a jboss-as-ejb-remote-app.jar, so the
module name is
        // jboss-as-ejb-remote-app
        final String moduleName = "jboss-as-ejb-remote-app";
        // AS7 allows each deployment to have an (optional) distinct name. We haven't specified
a distinct name for
        // our EJB deployment, so this is an empty string
        final String distinctName = "";
        // The EJB name which by default is the simple class name of the bean implementation
class
        final String beanName = CounterBean.class.getSimpleName();
        // the remote view fully qualified class name
        final String viewClassName = RemoteCounter.class.getName();
        // let's do the lookup (notice the ?stateful string as the last part of the jndi name
for stateful bean lookup)
        return (RemoteCounter) context.lookup("ejb:" + appName + "/" + moduleName + "/" +
distinctName + "/" + beanName + "!" + viewClassName + "?stateful");
    }
}
```

The entire server side and client side code is hosted at the github repo here ejb-remote

The code has some comments which will help you understand each of those lines. But we'll explain here in more detail what the code does. As a first step in the client code, we'll do a lookup of the EJB using a JNDI name. In AS7, for remote access to EJBs, you use the ejb: namespace with the following syntax:

For stateless beans:

```
ejb:<app-name>/<module-name>/<distinct-name>/<bean-name>!<fully-qualified-classname-of-the-remote-
```

For stateful beans:

```
ejb:<app-name>/<module-name>/<distinct-name>/<bean-name>!<fully-qualified-classname-of-the-remote-
```

The ejb: namespace identifies it as a EJB lookup and is a constant (i.e. doesn't change) for doing EJB lookups. The rest of the parts in the jndi name are as follows:

app-name : This is the name of the .ear (without the .ear suffix) that you have deployed on the server and contains your EJBs.

- Java EE 6 allows you to override the application name, to a name of your choice by setting it in the application.xml. If the deployment uses uses such an override then the app-name used in the JNDI name should match that name.
- EJBs can also be deployed in a .war or a plain .jar (like we did in step 1). In such cases where the deployment isn't an .ear file, then the app-name must be an empty string, while doing the lookup.

module-name : This is the name of the .jar (without the .jar suffix) that you have deployed on the server and the contains your EJBs. If the EJBs are deployed in a .war then the module name is the .war name (without the .war suffix).

- Java EE 6 allows you to override the module name, by setting it in the ejb-jar.xml/web.xml of your deployment. If the deployment uses such an override then the module-name used in the JNDI name should match that name.
- Module name part cannot be an empty string in the JNDI name

distinct-name : This is a WildFly-specific name which can be optionally assigned to the deployments that are deployed on the server. More about the purpose and usage of this will be explained in a separate chapter. If a deployment doesn't use distinct-name then, use an empty string in the JNDI name, for distinct-name

bean-name : This is the name of the bean for which you are doing the lookup. The bean name is typically the unqualified classname of the bean implementation class, but can be overriden through either ejb-jar.xml or via annotations. The bean name part cannot be an empty string in the JNDI name.

fully-qualified-classname-of-the-remote-interface : This is the fully qualified class name of the interface for which you are doing the lookup. The interface should be one of the remote interfaces exposed by the bean on the server. The fully qualified class name part cannot be an empty string in the JNDI name.

For stateful beans, the JNDI name expects an additional "?stateful" to be appended after the fully qualified interface name part. This is because for stateful beans, a new session gets created on JNDI lookup and the EJB client API implementation doesn't contact the server during the JNDI lookup to know what kind of a bean the JNDI name represents (we'll come to this in a while). So the JNDI name itself is expected to indicate that the client is looking up a stateful bean, so that an appropriate session can be created.

Now that we know the syntax, let's see our code and check what JNDI name it uses. Since our stateless EJB named CalculatorBean is deployed in a jboss-as-ejb-remote-app.jar (without any ear) and since we are looking up the org.jboss.as.quickstarts.ejb.remote.stateless.RemoteCalculator remote interface, our JNDI name will be:

```
ejb:/jboss-as-ejb-remote-app//CalculatorBean!org.jboss.as.quickstarts.ejb.remote.stateless.RemoteC
```

That's what the lookupRemoteStatelessCalculator() method in the above client code uses.

For the stateful EJB named CounterBean which is deployed in hte same jboss-as-ejb-remote-app.jar and which exposes the org.jboss.as.quickstarts.ejb.remote.stateful.RemoteCounter, the JNDI name will be:

```
ejb:/jboss-as-ejb-remote-app//CounterBean!org.jboss.as.quickstarts.ejb.remote.stateful.RemoteCount
```

That's what the lookupRemoteStatefulCounter() method in the above client code uses.

Now that we know of the JNDI name, let's take a look at the following piece of code in the lookupRemoteStatelessCalculator():

```
final Hashtable jndiProperties = new Hashtable();
jndiProperties.put(Context.URL_PKG_PREFIXES, "org.jboss.ejb.client.naming");
final Context context = new InitialContext(jndiProperties);
```

Here we are creating a JNDI InitialContext object by passing it some JNDI properties. The Context.URL_PKG_PREFIXES is set to org.jboss.ejb.client.naming. This is necessary because we should let the JNDI API know what handles the ejb: namespace that we use in our JNDI names for lookup. The "org.jboss.ejb.client.naming" has a URLContextFactory implementation which will be used by the JNDI APIs to parse and return an object for ejb: namespace lookups. You can either pass these properties to the constructor of the InitialContext class or have a jndi.properites file in the classpath of the client application, which (atleast) contains the following property:

```
java.naming.factory.url.pkgs=org.jboss.ejb.client.naming
```

So at this point, we have setup the InitialContext and also have the JNDI name ready to do the lookup. You can now do the lookup and the appropriate proxy which will be castable to the remote interface that you used as the fully qualified class name in the JNDI name, will be returned. Some of you might be wondering, how the JNDI implementation knew which server address to look, for your deployed EJBs. The answer is in AS7, the proxies returned via JNDI name lookup for ejb: namespace do not connect to the server unless an invocation on those proxies is done.

Now let's get to the point where we invoke on this returned proxy:

```
// Let's lookup the remote stateless calculator
        final RemoteCalculator statelessRemoteCalculator = lookupRemoteStatelessCalculator();
        System.out.println("Obtained a remote stateless calculator for invocation");
        // invoke on the remote calculator
        int a = 204;
        int b = 340;
        System.out.println("Adding " + a + " and " + b + " via the remote stateless calculator
deployed on the server");
        int sum = statelessRemoteCalculator.add(a, b);
```

We can see here that the proxy returned after the lookup is used to invoke the add(...) method of the bean. It's at this point that the JNDI implementation (which is backed by the EJB client API) needs to know the server details. So let's now get to the important part of setting up the EJB client context properties.

6.21.3 Setting up EJB client context properties

A EJB client context is a context which contains contextual information for carrying out remote invocations on EJBs. This is a WildFly-specific API. The EJB client context can be associated with multiple EJB receivers. Each EJB receiver is capable of handling invocations on different EJBs. For example, an EJB receiver "Foo" might be able to handle invocation on a bean identified by app-A/module-A/distinctinctName-A/Bar!RemoteBar, whereas a EJB receiver named "Blah" might be able to handle invocation on a bean identified by app-B/module-B/distinctName-B/BeanB!RemoteBean. Each such EJB receiver knows about what set of EJBs it can handle and each of the EJB receiver knows which server target to use for handling the invocations on the bean. For example, if you have a AS7 server at 10.20.30.40 IP address which has its remoting port opened at 4447 and if that's the server on which you deployed that CalculatorBean, then you can setup a EJB receiver which knows its target address is 10.20.30.40:4447. Such an EJB receiver will be capable enough to communicate to the server via the JBoss specific EJB remote client protocol (details of which will be explained in-depth in a separate chapter).

Now that we know what a EJB client context is and what a EJB receiver is, let's see how we can setup a client context with 1 EJB receiver which can connect to 10.20.30.40 IP address at port 4447. That EJB client context will then be used (internally) by the JNDI implementation to handle invocations on the bean proxy.

The client will have to place a jboss-ejb-client.properties file in the classpath of the application. The jboss-ejb-client.properties can contain the following properties:

```
endpoint.name=client-endpoint
remote.connectionprovider.create.options.org.xnio.Options.SSL_ENABLED=false

remote.connections=default

remote.connection.default.host=10.20.30.40
remote.connection.default.port = 8080
remote.connection.default.connect.options.org.xnio.Options.SASL_POLICY_NOANONYMOUS=false

remote.connection.default.username=appuser
remote.connection.default.password=apppassword
```

This file includes a reference to a default password. Be sure to change this as soon as possible.

The above properties file is just an example. The actual file that was used for this sample program is available here for reference jboss-ejb-client.properties

We'll see what each of it means.

First the endpoint.name property. We mentioned earlier that the EJB receivers will communicate with the server for EJB invocations. Internally, they use JBoss Remoting project to carry out the communication. The endpoint.name property represents the name that will be used to create the client side of the enpdoint. The endpoint.name property is optional and if not specified in the jboss-ejb-client.properties file, it will default to "config-based-ejb-client-endpoint" name.

Next is the remote.connectionprovider.create.options.<....> properties:

```
remote.connectionprovider.create.options.org.xnio.Options.SSL_ENABLED=false
```

The "remote.connectionprovider.create.options." property prefix can be used to pass the options that will be used while create the connection provider which will handle the "remote:" protocol. In this example we use the "remote.connectionprovider.create.options." property prefix to pass the "org.xnio.Options.SSL_ENABLED" property value as false. That property will then be used during the connection provider creation. Similarly other properties can be passed too, just append it to the "remote.connectionprovider.create.options." prefix

Next we'll see:

```
remote.connections=default
```

This is where you define the connections that you want to setup for communication with the remote server. The "remote.connections" property uses a comma separated value of connection "names". The connection names are just logical and are used grouping together the connection configuration properties later on in the properties file. The example above sets up a single remote connection named "default". There can be more than one connections that are configured. For example:

```
remote.connections=one, two
```

Here we are listing 2 connections named "one" and "two". Ultimately, each of the connections will map to a EJB receiver. So if you have 2 connections, that will setup 2 EJB receivers that will be added to the EJB client context. Each of these connections will be configured with the connection specific properties as follows:

```
remote.connection.default.host=10.20.30.40
remote.connection.default.port = 8080
remote.connection.default.connect.options.org.xnio.Options.SASL_POLICY_NOANONYMOUS=false
```

As you can see we are using the "remote.connection.<connection-name>." prefix for specifying the connection specific property. The connection name here is "default" and we are setting the "host" property of that connection to point to 10.20.30.40. Similarly we set the "port" for that connection to 4447.

By default WildFly uses 8080 as the remoting port. The EJB client API uses the http port, with the http-upgrade functionality, for communicating with the server for remote invocations, so that's the port we use in our client programs (unless the server is configured for some other http port)

```
remote.connection.default.username=appuser
remote.connection.default.password=apppassword
```

The given user/password must be set by using the command bin/add-user.sh (or.bat).
The user and password must be set because the security-realm is enabled for the subsystem remoting (see standalone*.xml or domain.xml) by default.
If you do not need the security for remoting you might remove the attribute security-realm in the configuration.

security-realm is enabled by default.

We then use the "remote.connection.<connection-name>.connect.options." property prefix to setup options that will be used during the connection creation.

Here's an example of setting up multiple connections with different properties for each of those:

```
remote.connectionprovider.create.options.org.xnio.Options.SSL_ENABLED=false

remote.connections=one, two

remote.connection.one.host=localhost
remote.connection.one.port=6999
remote.connection.one.connect.options.org.xnio.Options.SASL_POLICY_NOANONYMOUS=false

remote.connection.two.host=localhost
remote.connection.two.port=7999
remote.connection.two.connect.options.org.xnio.Options.SASL_POLICY_NOANONYMOUS=false
```

As you can see we setup 2 connections "one" and "two" which both point to "localhost" as the "host" but different ports. Each of these connections will internally be used to create the EJB receivers in the EJB client context.

So that's how the jboss-ejb-client.properties file can be setup and placed in the classpath.

Using a different file for setting up EJB client context

The EJB client code will by default look for jboss-ejb-client.properties in the classpath. However, you can specify a different file of your choice by setting the "jboss.ejb.client.properties.file.path" system property which points to a properties file on your filesystem, containing the client context configurations. An example for that would be "-Djboss.ejb.client.properties.file.path=/home/me/my-client/custom-jboss-ejb-client.properties"

Setting up the client classpath with the jars that are required to run the client application

A jboss-client jar is shipped in the distribution. It's available at WILDFLY_HOME/bin/client/jboss-client.jar. Place this jar in the classpath of your client application.

If you are using Maven to build the client application, then please follow the instructions in the WILDFLY_HOME/bin/client/README.txt to add this jar as a Maven dependency.

6.21.4 Summary

In the above examples, we saw what it takes to invoke a EJB from a remote client. To summarize:

- On the server side you need to deploy EJBs which expose the remote views.
- On the client side you need a client program which:
  - Has a jboss-ejb-client.properties in its classpath to setup the server connection information
  - Either has a jndi.properties to specify the java.naming.factory.url.pkgs property or passes that as a property to the InitialContext constructor
  - Setup the client classpath to include the jboss-client jar that's required for remote invocation of the EJBs. The location of the jar is mentioned above. You'll also need to have your application's bean interface jars and other jars that are required by your application, in the client classpath

6.22 EJB invocations from a remote server instance

The purpose of this chapter is to demonstrate how to lookup and invoke on EJBs deployed on an WildFly server instance **from another** WildFly server instance. This is different from invoking the EJBs from a remote standalone client

Let's call the server, from which the invocation happens to the EJB, as "Client Server" and the server on which the bean is deployed as the "Destination Server".

Note that this chapter deals with the case where the bean is deployed on the "Destination Server" but **not** on the "Client Server".

6.22.1 Application packaging

In this example, we'll consider a EJB which is packaged in a myejb.jar which is within a myapp.ear. Here's how it would look like:

```
myapp.ear
|
|---- myejb.jar
|        |
|        |---- <org.myapp.ejb.*> // EJB classes
```

Note that packaging itself isn't really important in the context of this article. You can deploy the EJBs in any standard way (.ear, .war or .jar).

6.22.2 Beans

In our example, we'll consider a simple stateless session bean which is as follows:

```
package org.myapp.ejb;

public interface Greeter {

    String greet(String user);
}
```

```
package org.myapp.ejb;

import javax.ejb.Remote;
import javax.ejb.Stateless;

@Stateless
@Remote (Greeter.class)
public class GreeterBean implements Greeter {

    @Override
    public String greet(String user) {
        return "Hello " + user + ", have a pleasant day!";
    }
}
```

6.22.3 Security

WildFly 8 is secure by default. What this means is that no communication can happen with an WildFly instance from a remote client (irrespective of whether it is a standalone client or another server instance) without passing the appropriate credentials. Remember that in this example, our "client server" will be communicating with the "destination server". So in order to allow this communication to happen successfully, we'll have to configure user credentials which we will be using during this communication. So let's start with the necessary configurations for this.

6.22.4 Configuring a user on the "Destination Server"

As a first step we'll configure a user on the destination server who will be allowed to access the destination server. We create the user using the `add-user` script that's available in the JBOSS_HOME/bin folder. In this example, we'll be configuring a `Application User` named `ejb` and with a password `test` in the `ApplicationRealm`. Running the `add-user` script is an interactive process and you will see questions/output as follows:

add-user

```
jpai@jpai-laptop:bin$ ./add-user.sh

What type of user do you wish to add?
 a) Management User (mgmt-users.properties)
 b) Application User (application-users.properties)
(a): b

Enter the details of the new user to add.
Realm (ApplicationRealm) :
Username : ejb
Password :
Re-enter Password :
What roles do you want this user to belong to? (Please enter a comma separated list, or leave
blank for none)\[  \]:
About to add user 'ejb' for realm 'ApplicationRealm'
Is this correct yes/no? yes
Added user 'ejb' to file
'/jboss-as-7.1.1.Final/standalone/configuration/application-users.properties'
Added user 'ejb' to file
'/jboss-as-7.1.1.Final/domain/configuration/application-users.properties'
Added user 'ejb' with roles to file
'/jboss-as-7.1.1.Final/standalone/configuration/application-roles.properties'
Added user 'ejb' with roles to file
'/jboss-as-7.1.1.Final/domain/configuration/application-roles.properties'
```

As you can see in the output above we have now configured a user on the destination server who'll be allowed to access this server. We'll use this user credentials later on in the client server for communicating with this server. The important bits to remember are the user we have created in this example is `ejb` and the password is `test.`

Note that you can use any username and password combination you want to.

You do **not** require the server to be started to add a user using the add-user script.

6.22.5 Start the "Destination Server"

As a next step towards running this example, we'll start the "Destination Server". In this example, we'll use the standalone server and use the *standalone-full.xml* configuration. The startup command will look like:

```
./standalone.sh -server-config=standalone-full.xml
```

Ensure that the server has started without any errors.

It's very important to note that if you are starting both the server instances on the same machine, then each of those server instances **must** have a unique `jboss.node.name` system property. You can do that by passing an appropriate value for `-Djboss.node.name` system property to the startup script:

```
./standalone.sh -server-config=standalone-full.xml -Djboss.node.name=<add appropriate
value here>
```

6.22.6 Deploying the application

The application (*myapp.ear* in our case) will be deployed to "Destination Server". The process of deploying the application is out of scope of this chapter. You can either use the Command Line Interface or the Admin console or any IDE or manually copy it to JBOSS_HOME/standalone/deployments folder (for standalone server). Just ensure that the application has been deployed successfully.

So far, we have built a EJB application and deployed it on the "Destination Server". Now let's move to the "Client Server" which acts as the client for the deployed EJBs on the "Destination Server".

6.22.7 Configuring the "Client Server" to point to the EJB remoting connector on the "Destination Server"

As a first step on the "Client Server", we need to let the server know about the "Destination Server"'s EJB remoting connector, over which it can communicate during the EJB invocations. To do that, we'll have to add a "*remote-outbound-connection*" to the remoting subsystem on the "Client Server". The "*remote-outbound-connection*" configuration indicates that a outbound connection will be created to a remote server instance from that server. The "*remote-outbound-connection*" will be backed by a "*outbound-socket-binding*" which will point to a remote host and a remote port (of the "Destination Server"). So let's see how we create these configurations.

6.22.8 Start the "Client Server"

In this example, we'll start the "Client Server" on the same machine as the "Destination Server". We have copied the entire server installation to a different folder and while starting the "Client Server" we'll use a port-offset (of 100 in this example) to avoid port conflicts:

```
./standalone.sh -server-config=standalone-full.xml -Djboss.socket.binding.port-offset=100
```

6.22.9 Create a security realm on the client server

Remember that we need to communicate with a secure destination server. In order to do that the client server has to pass the user credentials to the destination server. Earlier we created a user on the destination server who'll be allowed to communicate with that server. Now on the "client server" we'll create a security-realm which will be used to pass the user information.

In this example we'll use a security realm which stores a Base64 encoded password and then passes on that credentials when asked for. Earlier we created a user named `ejb` and password `test`. So our first task here would be to create the base64 encoded version of the password `test`. You can use any utility which generates you a base64 version for a string. I used this online site which generates the base64 encoded string. So for the `test` password, the base64 encoded version is `dGVzdA==`

While generating the base64 encoded string make sure that you don't have any trailing or leading spaces for the original password. That can lead to incorrect encoded versions being generated.

With new versions the add-user script will show the base64 password if you type 'y' if you've been ask

```
Is this new user going to be used for one AS process to connect to another AS process
e.g. slave domain controller?
```

Now that we have generated that base64 encoded password, let's use in the in the security realm that we are going to configure on the "client server". I'll first shutdown the client server and edit the standalone-full.xml file to add the following in the `<management>` section

Now let's create a "*security-realm*" for the base64 encoded password.

```
/core-service=management/security-realm=ejb-security-realm:add()
/core-service=management/security-realm=ejb-security-realm/server-identity=secret:add(value=dGVzdA
```

Notice that the CLI show the message *"process-state" => "reload-required"*, so you have to restart the server before you can use this change.

upon successful invocation of this command, the following configuration will be created in the *management* section:

standalone-full.xml

```
<management>
        <security-realms>
            ...
            <security-realm name="ejb-security-realm">
                <server-identities>
                    <secret value="dGVzdA==" />
                </server-identities>
            </security-realm>
        </security-realms>
...
```

As you can see I have created a security realm named "ejb-security-realm" (you can name it anything) with the base64 encoded password. So that completes the security realm configuration for the client server. Now let's move on to the next step.

6.22.10 Create a outbound-socket-binding on the "Client Server"

Let's first create a *outbound-socket-binding* which points the "Destination Server"'s host and port. We'll use the CLI to create this configuration:

```
/socket-binding-group=standard-sockets/remote-destination-outbound-socket-binding=remote-ejb:add(h
port=8080)
```

The above command will create a outbound-socket-binding named "*remote-ejb*" (we can name it anything) which points to "localhost" as the host and port 8080 as the destination port. Note that the host information should match the host/IP of the "Destination Server" (in this example we are running on the same machine so we use "localhost") and the port information should match the http-remoting connector port used by the EJB subsystem (by default it's 8080). When this command is run successfully, we'll see that the standalone-full.xml (the file which we used to start the server) was updated with the following outbound-socket-binding in the socket-binding-group:

```
<socket-binding-group name="standard-sockets" default-interface="public"
port-offset="${jboss.socket.binding.port-offset:0}">
        ...
        <outbound-socket-binding name="remote-ejb">
            <remote-destination host="localhost" port="8080"/>
        </outbound-socket-binding>
    </socket-binding-group>
```

6.22.11 Create a "remote-outbound-connection" which uses this newly created "outbound-socket-binding"

Now let's create a "*remote-outbound-connection*" which will use the newly created outbound-socket-binding (pointing to the EJB remoting connector of the "Destination Server"). We'll continue to use the CLI to create this configuration:

```
/subsystem=remoting/remote-outbound-connection=remote-ejb-connection:add(outbound-socket-binding-r
protocol=http-remoting, security-realm=ejb-security-realm, username=ejb)
```

The above command creates a remote-outbound-connection, named "*remote-ejb-connection*" (we can name it anything), in the remoting subsystem and uses the previously created "*remote-ejb*" outbound-socket-binding (notice the outbound-socket-binding-ref in that command) with the http-remoting protocol. Furthermore, we also set the security-realm attribute to point to the security-realm that we created in the previous step. Also notice that we have set the username attribute to use the user name who is allowed to communicate with the destination server.

What this step does is, it creates a outbound connection, on the client server, to the remote destination server and sets up the username to the user who allowed to communicate with that destination server and also sets up the security-realm to a pre-configured security-realm capable of passing along the user credentials (in this case the password). This way when a connection has to be established from the client server to the destination server, the connection creation logic will have the necessary security credentials to pass along and setup a successful secured connection.

Now let's run the following two operations to set some default connection creation options for the outbound connection:

```
/subsystem=remoting/remote-outbound-connection=remote-ejb-connection/property=SASL_POLICY_NOANONYM
```

```
/subsystem=remoting/remote-outbound-connection=remote-ejb-connection/property=SSL_ENABLED:add(valu
```

Ultimately, upon successful invocation of this command, the following configuration will be created in the remoting subsystem:

```
<subsystem xmlns="urn:jboss:domain:remoting:1.1">
....
            <outbound-connections>
                <remote-outbound-connection name="remote-ejb-connection"
outbound-socket-binding-ref="remote-ejb" protocol="http-remoting"
security-realm="ejb-security-realm" username="ejb">
                    <properties>
                        <property name="SASL_POLICY_NOANONYMOUS" value="false"/>
                        <property name="SSL_ENABLED" value="false"/>
                    </properties>
                </remote-outbound-connection>
            </outbound-connections>
        </subsystem>
```

From a server configuration point of view, that's all we need on the "Client Server". Our next step is to deploy an application on the "Client Server" which will invoke on the bean deployed on the "Destination Server".

6.22.12 Packaging the client application on the "Client Server"

Like on the "Destination Server", we'll use .ear packaging for the client application too. But like previously mentioned, that's not mandatory. You can even use a .war or .jar deployments. Here's how our client application packaging will look like:

```
client-app.ear
|
|--- META-INF
|        |
|        |--- jboss-ejb-client.xml
|
|--- web.war
|        |
|        |--- WEB-INF/classes
|        |        |
|        |        |---- <org.myapp.FooServlet> // classes in the web app
```

In the client application we'll use a servlet which invokes on the bean deployed on the "Destination Server". We can even invoke the bean on the "Destination Server" from a EJB on the "Client Server". The code remains the same (JNDI lookup, followed by invocation on the proxy). The important part to notice in this client application is the file *jboss-ejb-client.xml* which is packaged in the META-INF folder of a top level deployment (in this case our client-app.ear). This *jboss-ejb-client.xml* contains the EJB client configurations which will be used during the EJB invocations for finding the appropriate destinations (also known as, EJB receivers). The contents of the jboss-ejb-client.xml are explained next.

If your application is deployed as a top level .war deployment, then the jboss-ejb-client.xml is expected to be placed in .war/WEB-INF/ folder (i.e. the same location where you place any web.xml file).

6.22.13 Contents on jboss-ejb-client.xml

The jboss-ejb-client.xml will look like:

```
<jboss-ejb-client xmlns="urn:jboss:ejb-client:1.0">
    <client-context>
        <ejb-receivers>
            <remoting-ejb-receiver outbound-connection-ref="remote-ejb-connection"/>
        </ejb-receivers>
    </client-context>
</jboss-ejb-client>
```

You'll notice that we have configured the EJB client context (for this application) to use a remoting-ejb-receiver which points to our earlier created "*remote-outbound-connection*" named "*remote-ejb-connection*". This links the EJB client context to use the "*remote-ejb-connection*" which ultimately points to the EJB remoting connector on the "Destination Server".

6.22.14 Deploy the client application

Let's deploy the client application on the "Client Server". The process of deploying the application is out of scope, of this chapter. You can use either the CLI or the admin console or a IDE or deploy manually to JBOSS_HOME/standalone/deployments folder. Just ensure that the application is deployed successfully.

6.22.15 Client code invoking the bean

We mentioned that we'll be using a servlet to invoke on the bean, but the code to invoke the bean isn't servlet specific and can be used in other components (like EJB) too. So let's see how it looks like:

```
import javax.naming.Context;
import java.util.Hashtable;
import javax.naming.InitialContext;

...
public void invokeOnBean() {
        try {
            final Hashtable props = new Hashtable();
            // setup the ejb: namespace URL factory
            props.put(Context.URL_PKG_PREFIXES, "org.jboss.ejb.client.naming");
            // create the InitialContext
            final Context context = new javax.naming.InitialContext(props);

            // Lookup the Greeter bean using the ejb: namespace syntax which is explained here
https://docs.jboss.org/author/display/AS71/EJB+invocations+from+a+remote+client+using+JNDI
            final Greeter bean = (Greeter) context.lookup("ejb:" + "myapp" + "/" + "myejb" + "/"
+ "" + "/" + "GreeterBean" + "!" + org.myapp.ejb.Greeter.class.getName());

   // invoke on the bean
   final String greeting = bean.greet("Tom");

            System.out.println("Received greeting: " + greeting);

        } catch (Exception e) {
            throw new RuntimeException(e);
        }
}
```

That's it! The above code will invoke on the bean deployed on the "Destination Server" and return the result.

6.23 Example Applications - Migrated to WildFly

6.23.1 Example Applications Migrated from Previous Releases

The applications in this section were written for a previous version of the server but have been modified to run on WildFly 8. Changes were made to resolve issues that arose during deployment and runtime or to fix problems with application behaviour. Each example below documents the changes that were made to get the application to run successfully on WildFly.

Seam 2 JPA example

To see details on changes required to run this application on WildFly, see Seam 2 JPA example deployment on WildFly 8.

Seam 2 DVD Store example

For details on how to migrate this demo application, see Seam 2 DVD Store example on WildFly 8 on Marek Novotny's Blog.

Seam 2 Booking example

For details on how to migrate this demo application, see Seam 2 Booking example on WildFly 8 on Marek Novotny's Blog.

Seam 2 Booking - step-by-step migration of binaries

This document takes a somewhat different "brute force" approach. The idea is to deploy the binaries to WildFly, then see what issues you hit and learn how debug and resolve them. See Seam 2 Booking EAR Migration of Binaries - Step by Step.

jBPM-Console application

Kris Verlaenen migrated this application from AS 5 to WildFly 8. For details about this migration, see jBPM5 on WildFly on his Kris's Blog.

Order application used for performance testing

Andy Miller migrated this application from AS 5 to WildFly. For details about this migration, see Order Application Migration from EAP5.1 to WildFly.

Migrate example application

A step by step work through of issues, and their solutions, that might crop up when migrating applications to WildFly 8. See the following github project for details.

6.23.2 Example Applications Based on EE6

Applications in this section were designed and written specifically to use the features and functions of EE6.

- Quickstarts: A number of quickstart applications were written to demonstrate Java EE 6 and a few additional technologies. They provide small, specific, working examples that can be used as a reference for your own project. For more information about the quickstarts, see Get Started Developing Applications

6.23.3 Porting the Order Application from EAP 5.1 to WildFly 8

Andy Miller ported an example Order application that was used for performance testing from EAP 5.1 to WildFly 8. These are the notes he made during the migration process.

Overview of the application

The application is relatively simple. it contains three servlets, some stateless session beans, a stateful session bean, and some entities.

In addition to application code changes, modifications were made to the way the EAR was packaged. This is because WildFly removed support of some proprietary features that were available in EAP 5.1.

Summary of changes

Code Changes

Modify JNDI lookup code

Since this application was first written for EAP 4.2/4.3, which did not support EJB reference injection, the servlets were using pre-EE 5 methods for looking up stateless and stateful session bean interfaces. While migrating to WildFly, it seemed a good time to change the code to use the @EJB annotation, although this was not a required change.

The real difference is in the lookup name. WildFly only supports the new EE 6 portable JNDI names rather than the old EAR structure based names. The JNDI lookup code changed as follows:

Example of code in the EAP 5.1 version:

```
try {
    context = new InitialContext();
    distributionCenterManager = (DistributionCenterManager)
context.lookup("OrderManagerApp/DistributionCenterManagerBean/local");
} catch(Exception lookupError) {
    throw new ServletException("Couldn't find DistributionCenterManager bean", lookupError);
}
try {
    customerManager = (CustomerManager)
context.lookup("OrderManagerApp/CustomerManagerBean/local");
} catch(Exception lookupError) {
    throw new ServletException("Couldn't find CustomerManager bean", lookupError);
}

try {
    productManager = (ProductManager)
context.lookup("OrderManagerApp/ProductManagerBean/local");
} catch(Exception lookupError) {
    throw new ServletException("Couldn't find the ProductManager bean", lookupError);
}
```

Example of how this is now coded in WildFly:

```
@EJB(lookup="java:app/OrderManagerEJB/DistributionCenterManagerBean!services.ejb.DistributionCente
DistributionCenterManager distributionCenterManager;

@EJB(lookup="java:app/OrderManagerEJB/CustomerManagerBean!services.ejb.CustomerManager")
private CustomerManager customerManager;

@EJB(lookup="java:app/OrderManagerEJB/ProductManagerBean!services.ejb.ProductManager")
private ProductManager productManager;
```

In addition to the change to injection, which was supported in EAP 5.1.0, the lookup name changed from:

```
OrderManagerApp/DistributionCenterManagerBean/local
```

to:

```
java:app/OrderManagerEJB/DistributionCenterManagerBean!services.ejb.DistributionCenterManager
```

All the other beans were changed in a similar manner. They are now based on the portable JNDI names described in EE 6.

Modify logging code

The next major change was to logging within the application. The old version was using the commons logging infrastructure and Log4J that is bundled in the application server. Rather than bundling third-party logging, the application was modified to use the new WildFly Logging infrastructure.

The code changes themselves are rather trivial, as this example illustrates:

Old JBoss Commons Logging/Log4J:

```
private static Log log = LogFactory.getLog(CustomerManagerBean.class);
```

New WildFly Logging

```
private static Logger logger = Logger.getLogger(CustomerManagerBean.class.toString());
```

Old JBoss Commons Logging/Log4J:

```
if(log.isTraceEnabled()) {
    log.trace("Just flushed " + batchSize + " rows to the database.");
    log.trace("Total rows flushed is " + (i+1));
}
```

New WildFly Logging:

```
if(logger.isLoggable(Level.TRACE)) {
    logger.log(Level.TRACE, "Just flushed " + batchSize + " rows to the database.");
    logger.log(Level.TRACE, "Total rows flushed is " + (i+1));
}
```

In addition to the code changes made to use the new AS7 JBoss log manager module, you must add this dependency to the `MANIFEST.MF` file as follows:

```
Manifest-Version: 1.0
Dependencies: org.jboss.logmanager
```

Modify the code to use Infinispan for 2nd level cache

Jboss Cache has been replaced by Infinispan for 2nd level cache. This requires modification of the `persistence.xml` file.

This is what the file looked like in EAP 5.1:

```
<properties>
<property name="hibernate.cache.region.factory_class"
value="org.hibernate.cache.jbc2.JndiMultiplexedJBossCacheRegionFactory"/>
<property name="hibernate.cache.region.jbc2.cachefactory" value="java:CacheManager"/>
<property name="hibernate.cache.use_second_level_cache" value="true"/>
<property name="hibernate.cache.use_query_cache" value="false"/>
<property name="hibernate.cache.use_minimal_puts" value="true"/>
<property name="hibernate.cache.region.jbc2.cfg.entity" value="mvcc-entity"/>
<property name="hibernate.cache.region_prefix" value="services"/>
</properties>
```

This is how it was modified to use Infinispan for the same configuration:

```
<properties>
<property name="hibernate.cache.use_second_level_cache" value="true"/>
<property name="hibernate.cache.use_minimal_puts" value="true"/>
</properties>
<shared-cache-mode>ENABLE_SELECTIVE</shared-cache-mode>
```

Most of the properties are removed since they will default to the correct values for the second level cache. See "Using the Infinispan second level cache" for more details.

That was the extent of the code changes required to migrate the application to AS7.

EAR Packaging Changes

Due to modular class loading changes, the structure of the existing EAR failed to deploy successfully in WildFly.

The old structure of the EAR was as follows:

```
$ jar tf OrderManagerApp.ear
META-INF/MANIFEST.MF
META-INF/application.xml
OrderManagerWeb.war
OrderManagerEntities.jar
OrderManagerEJB.jar
META-INF/
```

In this structure, the entities and the `persistence.xml` were in one jar file, `OrderManagerEntities.jar`, and the stateless and stateful session beans were in another jar file, `OrderManagerEJB.jar`. This did not work due to modular class loading changes in WildFly. There are a couple of ways to resolve this issue:

1. Modify the class path in the `MANIFEST.MF`
2. Flatten the code and put all the beans in one JAR file.

The second approach was selected because it simplified the EAR structure:

```
$ jar tf OrderManagerApp.ear
META-INF/application.xml
OrderManagerWeb.war
OrderManagerEJB.jar
META-INF/
```

Since there is no longer an `OrderManagerEntities.jar` file, the `applcation.xml` file was modified to remove the entry.

An entry was added to the `MANIFEST.MF` file in the `OrderManagerWeb.war` to resolve another class loading issue resulting from the modification to use EJB reference injection in the servlets.

```
Manifest-Version: 1.0
Dependencies: org.jboss.logmanager
Class-Path: OrderManagerEJB.jar
```

The `Class-Path` entry tells the application to look in the `OrderManagerEJB.jar` file for the injected beans.

Summary

Although the existing EAR structure could have worked with additional modifications to the `MANIFEST.MF` file, this approach seemed more appealing because it simplified the structure while maintaining the web tier in its own WAR.

The source files for both versions is attached so you can view the changes that were made to the application.

6.23.4 Seam 2 Booking Application - Migration of Binaries from EAP5.1 to WildFly

This is a step-by-step how-to guide on porting the Seam Booking application binaries from EAP5.1 to WildFly 8. Although there are better approaches for migrating applications, the purpose of this document is to show the types of issues you might encounter when migrating an application and how to debug and resolve those issues.

For this example, the application EAR is deployed to the JBOSS_HOME/standalone/deployments directory with no changes other than extracting the archives so we can modify the XML files contained within them.

Step 1: Build and deploy the EAP5.1 version of the Seam Booking application

1. Build the EAR

```
cd /EAP5_HOME/jboss-eap5.1/seam/examples/booking
    ~/tools/apache-ant-1.8.2/bin/ant explode
```

2. Copy the EAR to the JBOSS_HOME deployments directory:

```
cp -r
EAP5_HOME/jboss-eap-5.1/seam/examples/booking/exploded-archives/jboss-seam-booking.ear
AS7_HOME/standalone/deployments/
    cp -r
EAP5_HOME/jboss-eap-5.1/seam/examples/booking/exploded-archives/jboss-seam-booking.war
AS7_HOME/standalone/deployments/jboss-seam.ear
    cp -r
EAP5_HOME/jboss-eap-5.1/seam/examples/booking/exploded-archives/jboss-seam-booking.jar
AS7_HOME/standalone/deployments/jboss-seam.ear
```

3. Start the WildFly server and check the log. You will see:

```
INFO [org.jboss.as.deployment] (DeploymentScanner-threads - 1) Found
jboss-seam-booking.ear in deployment directory. To trigger deployment create a file called
jboss-seam-booking.ear.dodeploy
```

4. Create an empty file with the name `jboss-seam-booking.ear.dodeploy` and copy it into the deployments directory. In the log, you will now see the following, indicating that it is deploying:

```
INFO [org.jboss.as.server.deployment] (MSC service thread 1-1) Starting deployment of
"jboss-seam-booking.ear"
    INFO [org.jboss.as.server.deployment] (MSC service thread 1-3) Starting deployment of
"jboss-seam-booking.jar"
    INFO [org.jboss.as.server.deployment] (MSC service thread 1-6) Starting deployment of
"jboss-seam.jar"
    INFO [org.jboss.as.server.deployment] (MSC service thread 1-2) Starting deployment of
"jboss-seam-booking.war"
```

At this point, you will first encounter your first deployment error. In the next section, we will step through each issue and how to debug and resolve it.

Step 2: Debug and resolve deployment errors and exceptions

First Issue: java.lang.ClassNotFoundException: javax.faces.FacesException

When you deploy the application, the log contains the following error:

```
ERROR \[org.jboss.msc.service.fail\] (MSC service thread 1-1) MSC00001: Failed to start service
jboss.deployment.subunit."jboss-seam-booking.ear"."jboss-seam-booking.war".POST_MODULE:
org.jboss.msc.service.StartException in service
jboss.deployment.subunit."jboss-seam-booking.ear"."jboss-seam-booking.war".POST_MODULE:
Failed to process phase POST_MODULE of subdeployment "jboss-seam-booking.war" of deployment
"jboss-seam-booking.ear"
    (.. additional logs removed ...)
Caused by: java.lang.ClassNotFoundException: javax.faces.FacesException from \[Module
"deployment.jboss-seam-booking.ear:main" from Service Module Loader\]
    at org.jboss.modules.ModuleClassLoader.findClass(ModuleClassLoader.java:191)
```

What it means:

The ClassNotFoundException indicates a missing dependency. In this case, it can not find the class `javax.faces.FacesException` and you need to explicitly add the dependency.

How to resolve it:

Find the module name for that class in the `AS7_HOME/modules` directory by looking for a path that matches the missing class. In this case, you will find 2 modules that match:

```
javax/faces/api/main
    javax/faces/api/1.2
```

Both modules have the same module name: “javax.faces.api” but one in the main directory is for JSF 2.0 and the one located in the 1.2 directory is for JSF 1.2. If there was only one module available, we could simply create a `MANIFEST.MF` file and added the module dependency. But in this case, we want to use the JSF 1.2 version and not the 2.0 version in main, so we need to be able to specify one and exclude the other. To do this, we create a `jboss-deployment-structure.xml` file in the EAR `META-INF/` directory that contains the following data:

```
<jboss-deployment-structure xmlns="urn:jboss:deployment-structure:1.0">
  <deployment>
     <dependencies>
       <module name="javax.faces.api" slot="1.2" export="true"/>
     </dependencies>
  </deployment>
  <sub-deployment name="jboss-seam-booking.war">
   <exclusions>
       <module name="javax.faces.api" slot="main"/>
     </exclusions>
     <dependencies>
       <module name="javax.faces.api" slot="1.2"/>
     </dependencies>
  </sub-deployment>
</jboss-deployment-structure>
```

In the "deployment" section, we add the dependency for the `javax.faces.api` for the JSF 1.2 module. We also add the dependency for the JSF 1.2 module in the sub-deployment section for the WAR and exclude the module for JSF 2.0.

Redeploy the application by deleting the `standalone/deployments/jboss-seam-booking.ear.failed` file and creating a blank `jboss-seam-booking.ear.dodeploy` file in the same directory.

Next Issue: java.lang.ClassNotFoundException: org.apache.commons.logging.Log

When you deploy the application, the log contains the following error:

```
ERROR [org.jboss.msc.service.fail] (MSC service thread 1-8) MSC00001: Failed to start service
jboss.deployment.unit."jboss-seam-booking.ear".INSTALL:
org.jboss.msc.service.StartException in service
jboss.deployment.unit."jboss-seam-booking.ear".INSTALL:
Failed to process phase INSTALL of deployment "jboss-seam-booking.ear"
    (.. additional logs removed ...)
Caused by: java.lang.ClassNotFoundException: org.apache.commons.logging.Log from [Module
"deployment.jboss-seam-booking.ear.jboss-seam-booking.war:main" from Service Module Loader]
```

What it means:

The ClassNotFoundException indicates a missing dependency. In this case, it can not find the class `org.apache.commons.logging.Log` and you need to explicitly add the dependency.

How to resolve it:

Find the module name for that class in the `JBOSS_HOME/modules/` directory by looking for a path that matches the missing class. In this case, you will find one module that matches the path `org/apache/commons/logging/`. The module name is “org.apache.commons.logging”.

Modify the `jboss-deployment-structure.xml` to add the module dependency to the deployment section of the file.

```
<module name="org.apache.commons.logging" export="true"/>
```

The `jboss-deployment-structure.xml` should now look like this:

```
<jboss-deployment-structure xmlns="urn:jboss:deployment-structure:1.0">
  <deployment>
     <dependencies>
       <module name="javax.faces.api" slot="1.2" export="true"/>
       <module name="org.apache.commons.logging" export="true"/>
     </dependencies>
  </deployment>
  <sub-deployment name="jboss-seam-booking.war">
   <exclusions>
       <module name="javax.faces.api" slot="main"/>
     </exclusions>
     <dependencies>
       <module name="javax.faces.api" slot="1.2"/>
     </dependencies>
  </sub-deployment>
</jboss-deployment-structure>
```

Redeploy the application by deleting the `standalone/deployments/jboss-seam-booking.ear.failed` file and creating a blank `jboss-seam-booking.ear.dodeploy file` in the same directory.

Next Issue: java.lang.ClassNotFoundException: org.dom4j.DocumentException

When you deploy the application, the log contains the following error:

```
ERROR [org.apache.catalina.core.ContainerBase.[jboss.web].[default-host].[/seam-booking]] (MSC
service thread 1-3) Exception sending context initialized event to listener instance of class
org.jboss.seam.servlet.SeamListener: java.lang.NoClassDefFoundError: org/dom4j/DocumentException
    (... additional logs removed ...)
Caused by: java.lang.ClassNotFoundException: org.dom4j.DocumentException from [Module
"deployment.jboss-seam-booking.ear.jboss-seam.jar:main" from Service Module Loader]
```

What it means:

Again, the ClassNotFoundException indicates a missing dependency. In this case, it can not find the class `org.dom4j.DocumentException`.

How to resolve it:

Find the module name in the `JBOSS_HOME/modules/` directory by looking for the `org/dom4j/DocumentException`. The module name is “org.dom4j”.

Modify the `jboss-deployment-structure.xml` to add the module dependency to the deployment section of the file.

```
<module name="org.dom4j" export="true"/>
```

The `jboss-deployment-structure.xml` file should now look like this:

```
<jboss-deployment-structure xmlns="urn:jboss:deployment-structure:1.0">
  <deployment>
     <dependencies>
       <module name="javax.faces.api" slot="1.2" export="true"/>
       <module name="org.apache.commons.logging" export="true"/>
           <module name="org.dom4j" export="true"/>
          </dependencies>
  </deployment>
  <sub-deployment name="jboss-seam-booking.war">
   <exclusions>
       <module name="javax.faces.api" slot="main"/>
     </exclusions>
     <dependencies>
       <module name="javax.faces.api" slot="1.2"/>
     </dependencies>
  </sub-deployment>
</jboss-deployment-structure>
```

Redeploy the application by deleting the `standalone/deployments/jboss-seam-booking.ear.failed` file and creating a blank `jboss-seam-booking.ear.dodeploy file` in the same directory.

Next Issue: java.lang.ClassNotFoundException: org.hibernate.validator.InvalidValue

When you deploy the application, the log contains the following error:

```
ERROR [org.apache.catalina.core.ContainerBase.[jboss.web].[default-host].[/seam-booking]] (MSC
service thread 1-6) Exception sending context initialized event to listener instance of class
org.jboss.seam.servlet.SeamListener: java.lang.RuntimeException: Could not create Component:
org.jboss.seam.international.statusMessages
    (... additional logs removed ...)
Caused by: java.lang.ClassNotFoundException: org.hibernate.validator.InvalidValue from [Module
"deployment.jboss-seam-booking.ear.jboss-seam.jar:main" from Service Module Loader]
```

What it means:

Again, the ClassNotFoundException indicates a missing dependency. In this case, it can not find the class `org.hibernate.validator.InvalidValue`.

How to resolve it:

There is a module “org.hibernate.validator”, but the JAR does not contain the `org.hibernate.validator.InvalidValue` class, so adding the module dependency will not resolve this issue.

In this case, the JAR containing the class was part of the EAP 5.1 deployment. We will look for the JAR that contains the missing class in the `EAP5_HOME/jboss-eap-5.1/seam/lib/` directory. To do this, open a console and type the following:

```
cd EAP5_HOME/jboss-eap-5.1/seam/lib
grep 'org.hibernate.validator.InvalidValue' `find . -name '*.jar'`
```

The result shows:

```
Binary file ./hibernate-validator.jar matches
Binary file ./test/hibernate-all.jar matches
```

In this case, we need to copy the `hibernate-validator.jar` to the `jboss-seam-booking.ear/lib/` directory:

```
cp EAP5_HOME/jboss-eap-5.1/seam/lib/hibernate-validator.jar jboss-seam-booking.ear/lib
```

Redeploy the application by deleting the `standalone/deployments/jboss-seam-booking.ear.failed` file and creating a blank `jboss-seam-booking.ear.dodeploy file` in the same directory.

Next Issue: java.lang.InstantiationException: org.jboss.seam.jsf.SeamApplicationFactory

When you deploy the application, the log contains the following error:

```
INFO  [javax.enterprise.resource.webcontainer.jsf.config] (MSC service thread 1-7) Unsanitized
stacktrace from failed start...: com.sun.faces.config.ConfigurationException: Factory
'javax.faces.application.ApplicationFactory' was not configured properly.
 at
com.sun.faces.config.processor.FactoryConfigProcessor.verifyFactoriesExist(FactoryConfigProcessor.
[jsf-impl-2.0.4-b09-jbossorg-4.jar:2.0.4-b09-jbossorg-4]
 (... additional logs removed ...)
Caused by: javax.faces.FacesException: org.jboss.seam.jsf.SeamApplicationFactory
 at javax.faces.FactoryFinder.getImplGivenPreviousImpl(FactoryFinder.java:606)
[jsf-api-1.2_13.jar:1.2_13-b01-FCS]
 (... additional logs removed ...)
 at
com.sun.faces.config.processor.FactoryConfigProcessor.verifyFactoriesExist(FactoryConfigProcessor.
[jsf-impl-2.0.4-b09-jbossorg-4.jar:2.0.4-b09-jbossorg-4]
 ... 11 more
Caused by: java.lang.InstantiationException: org.jboss.seam.jsf.SeamApplicationFactory
 at java.lang.Class.newInstance0(Class.java:340) [:1.6.0_25]
 at java.lang.Class.newInstance(Class.java:308) [:1.6.0_25]
 at javax.faces.FactoryFinder.getImplGivenPreviousImpl(FactoryFinder.java:604)
[jsf-api-1.2_13.jar:1.2_13-b01-FCS]
 ... 16 more
```

What it means:

The com.sun.faces.config.ConfigurationException and java.lang.InstantiationException indicate a dependency issue. In this case, it is not as obvious.

How to resolve it:

We need to find the module that contains the com.sun.faces classes. While there is no com.sun.faces module, there are are two com.sun.jsf-impl modules. A quick check of the jsf-impl-1.2_13.jar in the 1.2 directory shows it contains the com.sun.faces classes.

As we did with the javax.faces.FacesException ClassNotFoundException, we want to use the JSF 1.2 version and not the JSF 2.0 version in main, so we need to be able to specify one and exclude the other. We need to modify the jboss-deployment-structure.xml to add the module dependency to the deployment section of the file. We also need to add it to the WAR subdeployment and exclude the JSF 2.0 module. The file should now look like this:

```
<jboss-deployment-structure xmlns="urn:jboss:deployment-structure:1.0">
  <deployment>
     <dependencies>
       <module name="javax.faces.api" slot="1.2" export="true"/>
              <module name="com.sun.jsf-impl" slot="1.2" export="true"/>
       <module name="org.apache.commons.logging" export="true"/>
       <module name="org.dom4j" export="true"/>
     </dependencies>
  </deployment>
  <sub-deployment name="jboss-seam-booking.war">
   <exclusions>
       <module name="javax.faces.api" slot="main"/>
       <module name="com.sun.jsf-impl" slot="main"/>
     </exclusions>
     <dependencies>
       <module name="javax.faces.api" slot="1.2"/>
              <module name="com.sun.jsf-impl" slot="1.2"/>
     </dependencies>
  </sub-deployment>
</jboss-deployment-structure>
```

Redeploy the application by deleting the `standalone/deployments/jboss-seam-booking.ear.failed` file and creating a blank `jboss-seam-booking.ear.dodeploy file` in the same directory.

Next Issue: java.lang.ClassNotFoundException: org.apache.commons.collections.ArrayStack

When you deploy the application, the log contains the following error:

```
ERROR [org.apache.catalina.core.ContainerBase.[jboss.web].[default-host].[/seam-booking]] (MSC
service thread 1-1) Exception sending context initialized event to listener instance of class
com.sun.faces.config.ConfigureListener: java.lang.RuntimeException:
com.sun.faces.config.ConfigurationException: CONFIGURATION FAILED!
org.apache.commons.collections.ArrayStack from [Module "deployment.jboss-seam-booking.ear:main"
from Service Module Loader]
    (... additional logs removed ...)
Caused by: java.lang.ClassNotFoundException: org.apache.commons.collections.ArrayStack from
[Module "deployment.jboss-seam-booking.ear:main" from Service Module Loader]
```

What it means:

Again, the ClassNotFoundException indicates a missing dependency. In this case, it can not find the class `org.apache.commons.collections.ArrayStack`.

How to resolve it:

Find the module name in the `JBOSS_HOME/modules/` directory by looking for the `org/apache/commons/collections` path. The module name is “org.apache.commons.collections”.

Modify the `jboss-deployment-structure.xml` to add the module dependency to the deployment section of the file.

```
<module name="org.apache.commons.collections" export="true"/>
```

The `jboss-deployment-structure.xml` file should now look like this:

```
<jboss-deployment-structure xmlns="urn:jboss:deployment-structure:1.0">
  <deployment>
     <dependencies>
       <module name="javax.faces.api" slot="1.2" export="true"/>
              <module name="com.sun.jsf-impl" slot="1.2" export="true"/>
       <module name="org.apache.commons.logging" export="true"/>
       <module name="org.dom4j" export="true"/>
       <module name="org.apache.commons.collections" export="true"/>
   </dependencies>
  </deployment>
  <sub-deployment name="jboss-seam-booking.war">
   <exclusions>
       <module name="javax.faces.api" slot="main"/>
       <module name="com.sun.jsf-impl" slot="main"/>
     </exclusions>
     <dependencies>
       <module name="javax.faces.api" slot="1.2"/>
              <module name="com.sun.jsf-impl" slot="1.2"/>
     </dependencies>
  </sub-deployment>
</jboss-deployment-structure>
```

Redeploy the application by deleting the `standalone/deployments/jboss-seam-booking.ear.failed` file and creating a blank `jboss-seam-booking.ear.dodeploy file` in the same directory.

Next Issue: Services with missing/unavailable dependencies

When you deploy the application, the log contains the following error:

```
ERROR [org.jboss.as.deployment] (DeploymentScanner-threads - 2) {"Composite operation failed and
was rolled back. Steps that failed:" => {"Operation step-2" => {"Services with
missing/unavailable dependencies" =>
["jboss.deployment.subunit.\"jboss-seam-booking.ear\".\"jboss-seam-booking.jar\".component.Authent
missing [
jboss.naming.context.java.comp.jboss-seam-booking.\"jboss-seam-booking.jar\".AuthenticatorAction.\
]","jboss.deployment.subunit.\"jboss-seam-booking.ear\".\"jboss-seam-booking.jar\".component.Hotel
missing [
jboss.naming.context.java.comp.jboss-seam-booking.\"jboss-seam-booking.jar\".HotelSearchingAction.
]","
<... additional logs removed ...>
"jboss.deployment.subunit.\"jboss-seam-booking.ear\".\"jboss-seam-booking.jar\".component.BookingL
missing [
jboss.naming.context.java.comp.jboss-seam-booking.\"jboss-seam-booking.jar\".BookingListAction.\"e
]","jboss.persistenceunit.\"jboss-seam-booking.ear/jboss-seam-booking.jar#bookingDatabase\"
missing [ jboss.naming.context.java.bookingDatasource ]"]}}}
```

What it means:

When you get a “Services with missing/unavailable dependencies” error, look that the text within the brackets after “missing”.

In this case you see:

```
missing [
jboss.naming.context.java.comp.jboss-seam-booking.\"jboss-seam-booking.jar\".AuthenticatorAction.\
]
```

The “/em” indicates an Entity Manager and datasource issue.

How to resolve it:

In WildFly 8, datasource configuration has changed and needs to be defined in the `standalone/configuration/standalone.xml` file. Since WildFly ships with an example database that is already defined in the standalone.xml file, we will modify the `persistence.xml` file to use that example database. Looking in the `standalone.xml` file, you can see that the jndi-name for the example database is "java:jboss/datasources/ExampleDS".

Modify the `jboss-seam-booking.jar/META-INF/persistence.xml` file to comment the existing jta-data-source element and replace it as follows:

```
<!-- <jta-data-source>java:/bookingDatasource</jta-data-source> -->
<jta-data-source>java:jboss/datasources/ExampleDS</jta-data-source>
```

Redeploy the application by deleting the `standalone/deployments/jboss-seam-booking.ear.failed` file and creating a blank `jboss-seam-booking.ear.dodeploy file` in the same directory.

Next Issue: java.lang.ClassNotFoundException: org.hibernate.cache.HashtableCacheProvider

When you deploy the application, the log contains the following error:

```
ERROR [org.jboss.msc.service.fail] (MSC service thread 1-4) MSC00001: Failed to start service
jboss.persistenceunit."jboss-seam-booking.ear/jboss-seam-booking.jar#bookingDatabase":
org.jboss.msc.service.StartException in service
jboss.persistenceunit."jboss-seam-booking.ear/jboss-seam-booking.jar#bookingDatabase": Failed to
start service
 at org.jboss.msc.service.ServiceControllerImpl$StartTask.run(ServiceControllerImpl.java:1786)
 (... log messages removed ...)
Caused by: javax.persistence.PersistenceException: [PersistenceUnit: bookingDatabase] Unable to
build EntityManagerFactory
 at org.hibernate.ejb.Ejb3Configuration.buildEntityManagerFactory(Ejb3Configuration.java:903)
 {... log messages removed ...)
Caused by: org.hibernate.HibernateException: could not instantiate RegionFactory
[org.hibernate.cache.internal.bridge.RegionFactoryCacheProviderBridge]
 at org.hibernate.cfg.SettingsFactory.createRegionFactory(SettingsFactory.java:355)
 (... log messages removed ...)
Caused by: java.lang.reflect.InvocationTargetException
 at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method) [:1.6.0_25]
 (... log messages removed ...)
Caused by: org.hibernate.cache.CacheException: could not instantiate CacheProvider
[org.hibernate.cache.HashtableCacheProvider]
 at
org.hibernate.cache.internal.bridge.RegionFactoryCacheProviderBridge.<init>(RegionFactoryCacheProv
... 20 more
Caused by: java.lang.ClassNotFoundException: org.hibernate.cache.HashtableCacheProvider from
[Module "org.hibernate:main" from local module loader @12a3793 (roots:
/home/sgilda/tools/jboss7/modules)]
 at org.jboss.modules.ModuleClassLoader.findClass(ModuleClassLoader.java:191)
 (... log messages removed ...)
```

What it means:

The ClassNotFoundException indicates a missing dependency. In this case, it can not find the class org.hibernate.cache.HashtableCacheProvider.

How to resolve it:

There is no module for “org.hibernate.cache”. In this case, the JAR containing the class was part of the EAP 5.1 deployment. We will look for the JAR that contains the missing class in the `EAP5_HOME/jboss-eap-5.1/seam/lib/` directory.

To do this, open a console and type the following:

```
cd EAP5_HOME/jboss-eap-5.1/seam/lib
grep 'org.hibernate.validator.InvalidValue' `find . -name '*.jar'`
```

The result shows:

```
Binary file ./hibernate-core.jar matches
Binary file ./test/hibernate-all.jar matches
```

In this case, we need to copy the `hibernate-core.jar` to the `jboss-seam-booking.ear/lib/` directory:

```
cp EAP5_HOME/jboss-eap-5.1/seam/lib/hibernate-core.jar jboss-seam-booking.ear/lib
```

Redeploy the application by deleting the `standalone/deployments/jboss-seam-booking.ear.failed` file and creating a blank `jboss-seam-booking.ear.dodeploy file` in the same directory.

Next Issue: java.lang.ClassCastException: org.hibernate.cache.HashtableCacheProvider

When you deploy the application, the log contains the following error:

```
ERROR [org.jboss.msc.service.fail] (MSC service thread 1-2) MSC00001: Failed to start service
jboss.persistenceunit."jboss-seam-booking.ear/jboss-seam-booking.jar#bookingDatabase":
org.jboss.msc.service.StartException in service
jboss.persistenceunit."jboss-seam-booking.ear/jboss-seam-booking.jar#bookingDatabase": Failed to
start service
 at org.jboss.msc.service.ServiceControllerImpl$StartTask.run(ServiceControllerImpl.java:1786)
 (... log messages removed ...)
Caused by: javax.persistence.PersistenceException: [PersistenceUnit: bookingDatabase] Unable to
build EntityManagerFactory
 at org.hibernate.ejb.Ejb3Configuration.buildEntityManagerFactory(Ejb3Configuration.java:903)
 (... log messages removed ...)
Caused by: org.hibernate.HibernateException: could not instantiate RegionFactory
[org.hibernate.cache.internal.bridge.RegionFactoryCacheProviderBridge]
 at org.hibernate.cfg.SettingsFactory.createRegionFactory(SettingsFactory.java:355)
 (... log messages removed ...)
Caused by: java.lang.reflect.InvocationTargetException
 at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method) [:1.6.0_25]
 (... log messages removed ...)
Caused by: org.hibernate.cache.CacheException: could not instantiate CacheProvider
[org.hibernate.cache.HashtableCacheProvider]
 at
org.hibernate.cache.internal.bridge.RegionFactoryCacheProviderBridge.<init>(RegionFactoryCacheProv
... 20 more
Caused by: java.lang.ClassCastException: org.hibernate.cache.HashtableCacheProvider cannot be
cast to org.hibernate.cache.spi.CacheProvider
 at
org.hibernate.cache.internal.bridge.RegionFactoryCacheProviderBridge.<init>(RegionFactoryCacheProv
... 20 more
```

What it means:

A ClassCastException can be a result of many problems. If you look at this exception in the log, it appears the class org.hibernate.cache.HashtableCacheProvider extends org.hibernate.cache.spi.CacheProvider and is being loaded by a different class loader than the class it extends. The org.hibernate.cache.HashtableCacheProvider class is in in the hibernate-core.jar and is being loaded by the application class loader. The class it extends, org.hibernate.cache.spi.CacheProvider, is in the org/hibernate/main/hibernate-core-4.0.0.Beta1.jar and is implicitly loaded by that module.

This is not obvious, but due to changes in Hibernate 4, this problem is caused by a backward compatibility issue due moving the HashtableCacheProvider class into another package. This class was moved from the org.hibernate.cache package to the org.hibernate.cache.internal package. If you don't remove the hibernate.cache.provider_class property from the persistence.xml file, it will force the Seam application to bundle the old Hibernate libraries, resulting in ClassCastExceptions, In WildFly, you should move away from using HashtableCacheProvider and use Infinispan instead.

How to resolve it:

In WildFly, you need to comment out the hibernate.cache.provider_class property in the `jboss-seam-booking.jar/META-INF persistence.xml` file as follows:

```
<!-- <property name="hibernate.cache.provider_class"
value="org.hibernate.cache.HashtableCacheProvider"/> -->
```

Redeploy the application by deleting the `standalone/deployments/jboss-seam-booking.ear.failed` file and creating a blank `jboss-seam-booking.ear.dodeploy file` in the same directory.

No more issues: Deployment errors should be resolved

At this point, the application should deploy without errors, but when you access the URL “http://localhost:8080/seam-booking/” in a browser and attempt "Account Login", you will get a runtime error “The page isn't redirecting properly”. In the next section, we will step through each runtime issue and how to debug and resolve it.

Step 3: Debug and resolve runtime errors and exceptions

First Issue: javax.naming.NameNotFoundException: Name 'jboss-seam-booking' not found in context ''

The application deploys successfully, but when you access the URL “http://localhost:8080/seam-booking/” in a browser, you get “The page isn't redirecting properly” and the log contains the following error:

```
SEVERE [org.jboss.seam.jsf.SeamPhaseListener] (http--127.0.0.1-8080-1) swallowing exception:
java.lang.IllegalStateException: Could not start transaction
 at org.jboss.seam.jsf.SeamPhaseListener.begin(SeamPhaseListener.java:598) [jboss-seam.jar:]
 (... log messages removed ...)
Caused by: org.jboss.seam.InstantiationException: Could not instantiate Seam component:
org.jboss.seam.transaction.synchronizations
 at org.jboss.seam.Component.newInstance(Component.java:2170) [jboss-seam.jar:]
 (... log messages removed ...)
Caused by: javax.naming.NameNotFoundException: Name 'jboss-seam-booking' not found in context ''
 at org.jboss.as.naming.util.NamingUtils.nameNotFoundException(NamingUtils.java:109)
 (... log messages removed ...)
```

What it means:

A NameNotFoundException indications a JNDI naming issue. JNDI naming rules have changed in WildFly and we need to modify the lookup names to follow the new rules.

How to resolve it:

To debug this, look earlier in the server log trace to what JNDI binding were used. Looking at the server log we see this:

```
15:01:16,138 INFO
```

```
[org.jboss.as.ejb3.deployment.processors.EjbJndiBindingsDeploymentUnitProcessor] (MSC service
thread 1-1) JNDI bindings for session bean named RegisterAction in deployment unit subdeployment
"jboss-seam-booking.jar" of deployment "jboss-seam-booking.ear" are as follows:

 java:global/jboss-seam-booking/jboss-seam-booking.jar/RegisterAction!org.jboss.seam.example.booki
java:app/jboss-seam-booking.jar/RegisterAction!org.jboss.seam.example.booking.Register
 java:module/RegisterAction!org.jboss.seam.example.booking.Register
 java:global/jboss-seam-booking/jboss-seam-booking.jar/RegisterAction
 java:app/jboss-seam-booking.jar/RegisterAction
 java:module/RegisterAction

15:01:16,138 INFO
[org.jboss.as.ejb3.deployment.processors.EjbJndiBindingsDeploymentUnitProcessor] (MSC service
thread 1-1) JNDI bindings for session bean named BookingListAction in deployment unit
subdeployment "jboss-seam-booking.jar" of deployment "jboss-seam-booking.ear" are as follows:

 java:global/jboss-seam-booking/jboss-seam-booking.jar/BookingListAction!org.jboss.seam.example.bo
java:app/jboss-seam-booking.jar/BookingListAction!org.jboss.seam.example.booking.BookingList
 java:module/BookingListAction!org.jboss.seam.example.booking.BookingList
 java:global/jboss-seam-booking/jboss-seam-booking.jar/BookingListAction
 java:app/jboss-seam-booking.jar/BookingListAction
 java:module/BookingListAction

15:01:16,138 INFO
[org.jboss.as.ejb3.deployment.processors.EjbJndiBindingsDeploymentUnitProcessor] (MSC service
thread 1-1) JNDI bindings for session bean named HotelBookingAction in deployment unit
subdeployment "jboss-seam-booking.jar" of deployment "jboss-seam-booking.ear" are as follows:

 java:global/jboss-seam-booking/jboss-seam-booking.jar/HotelBookingAction!org.jboss.seam.example.b
java:app/jboss-seam-booking.jar/HotelBookingAction!org.jboss.seam.example.booking.HotelBooking
 java:module/HotelBookingAction!org.jboss.seam.example.booking.HotelBooking
 java:global/jboss-seam-booking/jboss-seam-booking.jar/HotelBookingAction
 java:app/jboss-seam-booking.jar/HotelBookingAction
 java:module/HotelBookingAction

15:01:16,138 INFO
[org.jboss.as.ejb3.deployment.processors.EjbJndiBindingsDeploymentUnitProcessor] (MSC service
thread 1-1) JNDI bindings for session bean named AuthenticatorAction in deployment unit
subdeployment "jboss-seam-booking.jar" of deployment "jboss-seam-booking.ear" are as follows:

 java:global/jboss-seam-booking/jboss-seam-booking.jar/AuthenticatorAction!org.jboss.seam.example.
java:app/jboss-seam-booking.jar/AuthenticatorAction!org.jboss.seam.example.booking.Authenticator
 java:module/AuthenticatorAction!org.jboss.seam.example.booking.Authenticator
 java:global/jboss-seam-booking/jboss-seam-booking.jar/AuthenticatorAction
 java:app/jboss-seam-booking.jar/AuthenticatorAction
 java:module/AuthenticatorAction

15:01:16,139 INFO
[org.jboss.as.ejb3.deployment.processors.EjbJndiBindingsDeploymentUnitProcessor] (MSC service
thread 1-1) JNDI bindings for session bean named ChangePasswordAction in deployment unit
subdeployment "jboss-seam-booking.jar" of deployment "jboss-seam-booking.ear" are as follows:

 java:global/jboss-seam-booking/jboss-seam-booking.jar/ChangePasswordAction!org.jboss.seam.example
java:app/jboss-seam-booking.jar/ChangePasswordAction!org.jboss.seam.example.booking.ChangePassword
java:module/ChangePasswordAction!org.jboss.seam.example.booking.ChangePassword
 java:global/jboss-seam-booking/jboss-seam-booking.jar/ChangePasswordAction
 java:app/jboss-seam-booking.jar/ChangePasswordAction
 java:module/ChangePasswordAction
```

```
15:01:16,139 INFO
[org.jboss.as.ejb3.deployment.processors.EjbJndiBindingsDeploymentUnitProcessor] (MSC service
thread 1-1) JNDI bindings for session bean named HotelSearchingAction in deployment unit
subdeployment "jboss-seam-booking.jar" of deployment "jboss-seam-booking.ear" are as follows:

 java:global/jboss-seam-booking/jboss-seam-booking.jar/HotelSearchingAction!org.jboss.seam.example
java:app/jboss-seam-booking.jar/HotelSearchingAction!org.jboss.seam.example.booking.HotelSearching
java:module/HotelSearchingAction!org.jboss.seam.example.booking.HotelSearching
 java:global/jboss-seam-booking/jboss-seam-booking.jar/HotelSearchingAction
 java:app/jboss-seam-booking.jar/HotelSearchingAction
 java:module/HotelSearchingAction

15:01:16,140 INFO
[org.jboss.as.ejb3.deployment.processors.EjbJndiBindingsDeploymentUnitProcessor] (MSC service
thread 1-6) JNDI bindings for session bean named EjbSynchronizations in deployment unit
subdeployment "jboss-seam.jar" of deployment "jboss-seam-booking.ear" are as follows:

 java:global/jboss-seam-booking/jboss-seam/EjbSynchronizations!org.jboss.seam.transaction.LocalEjb
java:app/jboss-seam/EjbSynchronizations!org.jboss.seam.transaction.LocalEjbSynchronizations
 java:module/EjbSynchronizations!org.jboss.seam.transaction.LocalEjbSynchronizations
 java:global/jboss-seam-booking/jboss-seam/EjbSynchronizations
 java:app/jboss-seam/EjbSynchronizations
 java:module/EjbSynchronizations

15:01:16,140 INFO
[org.jboss.as.ejb3.deployment.processors.EjbJndiBindingsDeploymentUnitProcessor] (MSC service
thread 1-6) JNDI bindings for session bean named TimerServiceDispatcher in deployment unit
subdeployment "jboss-seam.jar" of deployment "jboss-seam-booking.ear" are as follows:

 java:global/jboss-seam-booking/jboss-seam/TimerServiceDispatcher!org.jboss.seam.async.LocalTimerS
java:app/jboss-seam/TimerServiceDispatcher!org.jboss.seam.async.LocalTimerServiceDispatcher
 java:module/TimerServiceDispatcher!org.jboss.seam.async.LocalTimerServiceDispatcher
 java:global/jboss-seam-booking/jboss-seam/TimerServiceDispatcher
 java:app/jboss-seam/TimerServiceDispatcher
 java:module/TimerServiceDispatcher
```

We need to modify the WAR's lib/components.xml file to use the new JNDI bindings. In the log, note the EJB JNDI bindings all start with "java:app/jboss-seam-booking.jar"

Replace the <core:init> element as follows:

```
<!--     <core:init jndi-pattern="jboss-seam-booking/#{ejbName}/local" debug="true"
distributable="false"/> -->
<core:init jndi-pattern="java:app/jboss-seam-booking.jar/#{ejbName}" debug="true"
distributable="false"/>
```

Next, we need to add the EjbSynchronizations and TimerServiceDispatcher JNDI bindings. Add the following component elements to the file:

```
<component class="org.jboss.seam.transaction.EjbSynchronizations"
jndi-name="java:app/jboss-seam/EjbSynchronizations"/>
<component class="org.jboss.seam.async.TimerServiceDispatcher"
jndi-name="java:app/jboss-seam/TimerServiceDispatcher"/>
```

The components.xml file should now look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:core="http://jboss.com/products/seam/core"
            xmlns:security="http://jboss.com/products/seam/security"
            xmlns:transaction="http://jboss.com/products/seam/transaction"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation=
                "http://jboss.com/products/seam/core http://jboss.com/products/seam/core-2.2.xsd
                 http://jboss.com/products/seam/transaction
http://jboss.com/products/seam/transaction-2.2.xsd
                 http://jboss.com/products/seam/security
http://jboss.com/products/seam/security-2.2.xsd
                 http://jboss.com/products/seam/components
http://jboss.com/products/seam/components-2.2.xsd">

    <!-- <core:init jndi-pattern="jboss-seam-booking/#{ejbName}/local" debug="true"
distributable="false"/> -->
    <core:init jndi-pattern="java:app/jboss-seam-booking.jar/#{ejbName}" debug="true"
distributable="false"/>

    <core:manager conversation-timeout="120000"
                  concurrent-request-timeout="500"
                  conversation-id-parameter="cid"/>

    <transaction:ejb-transaction/>

    <security:identity authenticate-method="#{authenticator.authenticate}"/>

    <component class="org.jboss.seam.transaction.EjbSynchronizations"
            jndi-name="java:app/jboss-seam/EjbSynchronizations"/>
    <component class="org.jboss.seam.async.TimerServiceDispatcher"
            jndi-name="java:app/jboss-seam/TimerServiceDispatcher"/>
</components>
```

Redeploy the application by deleting the `standalone/deployments/jboss-seam-booking.ear.failed` file and creating a blank `jboss-seam-booking.ear.dodeploy` file in the same directory.

At this point, the application should deploy and run without error. When you access the URL “http://localhost:8080/seam-booking/” in a browser, you will be able to login successfully.

Step 4: Access the application

Access the URL “http://localhost:8080/seam-booking/” in a browser and login with demo/demo. You should the Booking welcome page.

Summary of Changes

Although it would be much more efficient to determine dependencies in advance and add the implicit dependencies in one step, this exercise shows how problems appear in the log and provides some information on how to debug and resolve them.

The following is a summary of changes made to the application when migrating it to WildFly:

1. We created a `jboss-deployment-structure.xml` file in the EAR's `META-INF/` directory. We added "dependencies" and "exclusions" to resolve ClassNotFoundExceptions. This file contains the following data:

```
<jboss-deployment-structure xmlns="urn:jboss:deployment-structure:1.0">
  <deployment>
     <dependencies>
       <module name="javax.faces.api" slot="1.2" export="true"/>
       <module name="com.sun.jsf-impl" slot="1.2" export="true"/>
       <module name="org.apache.commons.logging" export="true"/>
           <module name="org.dom4j" export="true"/>
       <module name="org.apache.commons.collections" export="true"/>
     </dependencies>
  </deployment>
  <sub-deployment name="jboss-seam-booking.war">
   <exclusions>
       <module name="javax.faces.api" slot="main"/>
       <module name="com.sun.jsf-impl" slot="main"/>
     </exclusions>
     <dependencies>
       <module name="javax.faces.api" slot="1.2"/>
       <module name="com.sun.jsf-impl" slot="1.2"/>
     </dependencies>
  </sub-deployment>
</jboss-deployment-structure>
```

2. We copied the following JARs from the `EAP5_HOME/jboss-eap-5.1/seam/lib/` directory to the `jboss-seam-booking.ear/lib/` directory to resolve ClassNotFoundExceptions:

```
hibernate-core.jar
hibernate-validator.jar
```

3. We modified the {{jboss-seam-booking.jar/META-INF/persistence.xml} file as follows.
    1. First, we changed the jta-data-source element to use the Example database that ships with AS7:

```
<!-- <jta-data-source>java:/bookingDatasource</jta-data-source> -->
<jta-data-source>java:jboss/datasources/ExampleDS</jta-data-source>
```

    2. Next, we commented out the hibernate.cache.provider_class property:

```
<!-- <property name="hibernate.cache.provider_class"
value="org.hibernate.cache.HashtableCacheProvider"/> -->
```

4. We modified the WAR's lib/components.xml file to use the new JNDI bindings
    1. We replaced the <core:init> existing element as follows:

```
<!-- <core:init jndi-pattern="jboss-seam-booking/#{ejbName}/local" debug="true"
distributable="false"/> -->
<core:init jndi-pattern="java:app/jboss-seam-booking.jar/#{ejbName}" debug="true"
distributable="false"/>
```

    2. We added component elements for the "EjbSynchronizations" and "TimerServiceDispatcher" JNDI bindings

```
<component class="org.jboss.seam.transaction.EjbSynchronizations"
jndi-name="java:app/jboss-seam/EjbSynchronizations"/>
 <component class="org.jboss.seam.async.TimerServiceDispatcher"
jndi-name="java:app/jboss-seam/TimerServiceDispatcher"/>
```

The unmodified EAR from EAP 5.1 (jboss-seam-booking-eap51.ear.tar.gz) and the EAR as modified to run on AS7 (jboss-seam-booking-as7.ear.tar.gz) are attached to this document.

6.24 How do I migrate my application from AS7 to WildFly

6.24.1 About this Document

The purpose of this guide is to document changes that are needed to successfully run and deploy AS 7 applications on WildFly. It provides information on to resolve deployment and runtime problems and how to prevent changes in application behavior. This is the first step in moving to the new platform. Once the application is successfully deployed and running on the new platform, plans can be made to upgrade individual components to use the new functions and features of WildFly.

6.24.2 Overview of WildFly

The list of WildFly new functionality is extensive, being the most relevant, with respect to server and application migrations:

- Java EE7 - WildFly is a certified implementation of Java EE7, meeting both the Web and the Full profiles, and already includes support for the latest iterations of CDI (1.2) and Web Sockets (1.1).
- Undertow - A new cutting-edge web server in WildFly, designed for maximum throughput and scalability, including environments with over a million connections. And the latest web technologies, such as the new HTTP/2 standard, are already onboard.
- Apache ActiveMQ Artemis - WildFly's new JMS broker. Based on an code donation from HornetQ, this Apache subproject provides outstanding performance based on a proven non-blocking architecture.
- IronJacamar 1.2 - The latest IronJacamar provides a stable and feature rich JCA & Datasources support.
- JBossWS 5 - The fifth generation of JBossWS, a major leap forward, brings new features and performances improvements to WildFly Web Services
- RESTEasy 3 - WildFly includes the latest generation of RESTEasy, which goes beyond the standard Java EE REST APIs (JAX-RS 2.0), by also providing a number of useful extensions, such as JSON Web Encryption, Jackson, Yaml, JSON-P, and Jettison.
- OpenJDK ORB - WildFly switched the IIOP implementation from JacORB, to a downstream branch of the OpenJDK Orb, leading to better interoperability with the JVM ORB and the Java EE RI.
- Feature Rich Clustering - Clustering support was heavily refactored in WildFly, and includes several APIs for applications
- Port Reduction - By utilising HTTP upgrade, WildFly has moved nearly all of its protocols to be multiplexed over just two HTTP ports: a management port (9990), and an application port (8080).
- Enhanced Logging - The management API now supports the ability to list and view the available log files on a server, or even define custom formatters other than the default pattern formatter. Deployment's logging setup is also greatly enhanced.

The support for some technologies was removed, due to the high maintenance cost, low community interest, and much better alternative solutions:

- CMP EJB - JPA offers a much more performant and flexible API
- JAX-RPC - JAX-WS offers a much more accurate and complete solution
- JSR-88 - With very little adoption, the more complete deployment APIs provided by vendors are preferred

6.24.3 Server Migration

Migrating an AS7 server to WildFly consists of migrating custom configuration files, and some persisted data that may exist.

JacORB Subsystem

WildFly ORB support is provided by the JDK itself, instead of relying on JacORB. A subsystem configuration migration is required.

JacORB Subsystem Configuration

The extension's module **org.jboss.as.jacorb *is replaced by module *org.wildfly.iiop-openjdk**, while the subsystem configuration namespace **urn:jboss:domain:jacorb:2.0** is replaced by **urn:jboss:domain:iiop-openjdk:1.0**.

The XML configuration of the new subsystem accepts only a subset of the legacy elements/attributes. Consider the following example of the JacORB subsystem configuration, containing all valid elements and attributes:

```
<subsystem xmlns="urn:jboss:domain:jacorb:1.3">
   <orb name="JBoss" print-version="off" use-imr="off" use-bom="off"  cache-typecodes="off"
       cache-poa-names="off" giop-minor-version ="2" socket-binding="jacorb"
ssl-socket-binding="jacorb-ssl">
       <connection retries="5" retry-interval="500" client-timeout="0" server-timeout="0"
           max-server-connections="500" max-managed-buf-size="24" outbuf-size="2048"
           outbuf-cache-timeout="-1"/>
       <initializers security="off" transactions="spec"/>
   </orb>
   <poa monitoring="off" queue-wait="on" queue-min="10" queue-max="100">
       <request-processors pool-size="10" max-threads="32"/>
   </poa>
   <naming root-context="JBoss/Naming/root" export-corbaloc="on"/>
   <interop sun="on" comet="off" iona="off" chunk-custom-rmi-valuetypes="on"
       lax-boolean-encoding="off" indirection-encoding-disable="off"
strict-check-on-tc-creation="off"/>
   <security support-ssl="off" add-component-via-interceptor="on" client-supports="MutualAuth"
       client-requires="None" server-supports="MutualAuth" server-requires="None"/>
   <properties>
       <property name="some_property" value="some_value"/>
   </properties>
</subsystem>
```

Properties that are not supported and have to be removed:

- <orb/>: client-timeout, max-managed-buf-size, max-server-connections, outbuf-cache-timeout, outbuf-size, connection retries, retry-interval, name,server-timeout
- <poa/>: queue-min, queue-max, pool-size, max-threads

On-off properties: have to either be removed or in off mode:

- <orb/>: cache-poa-names, cache-typecodes, print-version, use-bom, use-imr
- <interop/>: all except sun
- <poa/>: monitoring, queue-wait

In case the legacy subsystem configuration is available, such configuration may be migrated to the new subsystem by invoking its `migrate` operation, using the CLI management client:

```
/subsystem=jacorb:migrate
```

There is also a `describe-migration` operation that returns a list of all the management operations that are performed to migrate from the legacy subsystem to the new one:

```
/subsystem=jacorb:describe-migration
```

Both `migrate` and `describe-migration` will also display a list of migration-warnings if there are some resource or attributes that can not be migrated automatically. The following is a list of these warnings:

- Properties X cannot be emulated using OpenJDK ORB and are not supported
  This warning means that mentioned properties are not supported and won't be included in the new subsystem configuration. As a result of that admin must be aware that any behaviour implied by those properties would be inexistent. Admin has to check whether subsystem is able to operate correctly without that behaviour on the new server.Unsupported properties: cache-poa-names, cache-typecodes, chunk-custom-rmi-valuetypes, client-timeout, comet, indirection-encoding-disable, iona, lax-boolean-encoding, max-managed-buf-size, max-server-connections, max-threads, outbuf-cache-timeout, outbuf-size, queue-max, queue-min, poa-monitoring, print-version, retries, retry-interval, queue-wait, server-timeout, strict-check-on-tc-creation, use-bom, use-imr.
- The properties X use expressions. Configuration properties that are used to resolve those expressions should be transformed manually to the new iiop-openjdk subsystem format
  Admin has to transform all the configuration files to work correctly with the jacorb subsystem. f.e. jacorb has a property giop-minor-version whereas openjdk uses property giop-version. Let's suppose we use '1' minor version in jacorb and have it configured in standalone.conf file as system variable: -Diiop-giop-minor-version=1. Admin is responsible for changing this variable to 1.1 after the migration to make sure that the new subsystem will work correctly.

JBoss Web Subsystem

JBoss Web is replaced by Undertow in WildFly, which means that the legacy subsystem configuration should be migrated to WildFly's Undertow subsystem configuration.

JBoss Web Subsystem Configuration

The extension's module **org.jboss.as.web *is replaced by module *org.wildfly.extension.undertow**, while the subsystem configuration namespace **urn:jboss:domain:web:*** is replaced by **urn:jboss:domain:undertow:3.0**.

The XML configuration of the new subsystem is relatively different. Consider the following example of the JBoss Web subsystem configuration, containing all valid elements and attributes:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<subsystem xmlns="urn:jboss:domain:web:2.2" default-virtual-server="default-host" native="true"
default-session-timeout="30" instance-id="foo">
    <configuration>
        <static-resources listings="true"
                          sendfile="1000"
                          file-encoding="utf-8"
                          read-only="true"
                          webdav="false"
                          secret="secret"
                          max-depth="5"
                          disabled="false"
                />
        <jsp-configuration development="true"
                           disabled="false"
                           keep-generated="true"
                           trim-spaces="true"
                           tag-pooling="true"
                           mapped-file="true"
                           check-interval="20"
                           modification-test-interval="1000"
                           recompile-on-fail="true"
                           smap="true"
                           dump-smap="true"
                           generate-strings-as-char-arrays="true"
                           error-on-use-bean-invalid-class-attribute="true"
                           scratch-dir="/some/dir"
                           source-vm="1.7"
                           target-vm="1.7"
                           java-encoding="utf-8"
                           x-powered-by="true"
                           display-source-fragment="true" />
        <mime-mapping name="ogx" value="application/ogg" />
        <welcome-file>titi</welcome-file>
    </configuration>
    <connector name="http" scheme="http"
               protocol="HTTP/1.1"
               socket-binding="http"
               enabled="true"
               enable-lookups="false"
               proxy-binding="reverse-proxy"
               max-post-size="2097153"
               max-save-post-size="512"
               redirect-binding="https"
               max-connections="300"
               secure="false"
               executor="some-executor"
            />
    <connector name="https" scheme="https" protocol="HTTP/1.1" secure="true"
socket-binding="https">
        <ssl certificate-key-file="${file-base}/server.keystore"
             ca-certificate-file="${file-base}/jsse.keystore"
             key-alias="test"
             password="changeit"
             cipher-suite="SSL_RSA_WITH_3DES_EDE_CBC_SHA"
             protocol="SSLv3"
             verify-client="true"
             verify-depth="3"
             certificate-file="certificate-file.ext"
```

```
            ca-revocation-url="https://example.org/some/url"
            ca-certificate-password="changeit"
            keystore-type="JKS"
            truststore-type="JKS"
            session-cache-size="512"
            session-timeout="3000"
            ssl-protocol="RFC4279"
               />
    </connector>
    <connector name="http-vs" scheme="http" protocol="HTTP/1.1" socket-binding="http" >
        <virtual-server name="vs1" />
        <virtual-server name="vs2" />
    </connector>
    <virtual-server name="default-host" enable-welcome-root="true" default-web-module="foo.war">
        <alias name="localhost" />
        <alias name="example.com" />
        <access-log resolve-hosts="true" extended="true" pattern="extended" prefix="prefix"
rotate="true" >
            <directory relative-to="jboss.server.base.dir" path="toto" />
        </access-log>
        <rewrite name="myrewrite" pattern="^/helloworld(.*)" substitution="/helloworld/test.jsp"
flags="L" />
        <rewrite name="with-conditions" pattern="^/helloworld(.*)"
substitution="/helloworld/test.jsp" flags="L" >
            <condition name="https" pattern="off" test="%{HTTPS}" flags="NC"/>
            <condition name="user" test="%{USER}" pattern="toto" flags="NC"/>
            <condition name="no-flags" test="%{USER}" pattern="toto"/>
        </rewrite>
        <sso reauthenticate="true" domain="myDomain" cache-name="myCache"
             cache-container="cache-container" http-only="true"/>
    </virtual-server>
    <virtual-server name="vs1" />
    <virtual-server name="vs2" />
    <valve name="myvalve" module="org.jboss.some.module" class-name="org.jboss.some.class"
enabled="true">
        <param param-name="param-name" param-value="some-value"/>
    </valve>
    <valve name="accessLog" module="org.jboss.as.web"
class-name="org.apache.catalina.valves.AccessLogValve">
        <param param-name="prefix" param-value="myapp_access_log." />
        <param param-name="suffix" param-value=".log" />
        <param param-name="rotatable" param-value="true" />
        <param param-name="fileDateFormat" param-value="yyyy-MM-dd" />
        <param param-name="pattern" param-value="common" />
        <param param-name="directory" param-value="${jboss.server.log.dir}" />
        <param param-name="resolveHosts" param-value="false"/>
        <param param-name="conditionIf" param-value="log-enabled"/>
    </valve>
    <valve name="request-dumper" module="org.jboss.as.web"
class-name="org.apache.catalina.valves.RequestDumperValve"/>
    <valve name="remote-addr" module="org.jboss.as.web"
class-name="org.apache.catalina.valves.RemoteAddrValve">
        <param param-name="allow" param-value="127.0.0.1,127.0.0.2" />
        <param param-name="deny" param-value="192.168.1.20" />
    </valve>
    <valve name="crawler" class-name="org.apache.catalina.valves.CrawlerSessionManagerValve"
module="org.jboss.as.web" >
        <param param-name="sessionInactiveInterval" param-value="1" />
```

```
        <param param-name="crawlerUserAgents" param-value="Google" />
    </valve>
    <valve name="proxy" class-name="org.apache.catalina.valves.RemoteIpValve"
module="org.jboss.as.web" >
        <param param-name="internalProxies" param-value="192\.168\.0\.10|192\.168\.0\.11" />
        <param param-name="remoteIpHeader" param-value="x-forwarded-for" />
        <param param-name="proxiesHeader" param-value="x-forwarded-by" />
        <param param-name="trustedProxies" param-value="proxy1|proxy2" />
    </valve>
</subsystem>
```

FIXME compare with Undertow, list unsupported features

It's possible to do a migration of the legacy subsystem configuration, and related persisted data. , by invoking the legacy's subsystem's `migrate` operation, using the CLI management client:

```
/subsystem=web:migrate
```

There is also a `describe-migration` operation that returns a list of all the management operations that are performed to migrate from the legacy subsystem to the new one:

```
/subsystem=web:describe-migration
```

Both `migrate` and `describe-migration` will also display a list of migration-warnings if there are some resource or attributes that can not be migrated automatically. The following is a list of these warnings:

- Could not migrate resource X
  This warning means that mentioned resource configuration is not supported and won't be included in the new subsystem configuration. As a result of that admin must be aware that any behaviour implied by those resources would be inexistent. Admin has to check whether subsystem is able to operate correctly without that behaviour on the new server.
  FIXME must document which are the resources that trigger this
- Could not migrate attribute X from resource Y.
  This warning means that mentioned resource configuration property is not supported and won't be included in the new subsystem configuration. As a result of that admin must be aware that any behaviour implied by those properties would be inexistent. Admin has to check whether subsystem is able to operate correctly without that behaviour on the new server.
  FIXME must document which are the properties that trigger this
- Could not migrate SSL connector as no SSL config is defined
- Could not migrate verify-client attribute %s to the Undertow equivalent
- Could not migrate verify-client expression %s
- Could not migrate valve X
  This warning means that mentioned valve configuration is not supported and won't be included in the new subsystem configuration. As a result of that admin must be aware that any behaviour implied by those resources would be inexistent. Admin has to check whether subsystem is able to operate correctly without that behaviour on the new server. This warning may happen for :
  - org.apache.catalina.valves.RemoteAddrValve : must have at least one allowed or denied value.
  - org.apache.catalina.valves.RemoteHostValve : must have at least one allowed or denied value.
  - org.apache.catalina.authenticator.BasicAuthenticator
  - org.apache.catalina.authenticator.DigestAuthenticator
  - org.apache.catalina.authenticator.FormAuthenticator
  - org.apache.catalina.authenticator.SSLAuthenticator
  - org.apache.catalina.authenticator.SpnegoAuthenticator
  - custom valves
- Could not migrate attribute X from valve Y
  This warning means that mentioned valve configuration property is not supported and won't be included in the new subsystem configuration. As a result of that admin must be aware that any behaviour implied by those properties would be inexistent. Admin has to check whether subsystem is able to operate correctly without that behaviour on the new server. This warning may happen for :
  - org.apache.catalina.valves.AccessLogValve : if you use the following parameters *resolveHosts* , *fileDateFormat*, *renameOnRotate*, *encoding*, *locale*, *requestAttributesEnabled*, *buffered*.
  - org.apache.catalina.valves.ExtendedAccessLogValve : if you use the following parameters *resolveHosts*, *fileDateFormat*, *renameOnRotate*, *encoding*, *locale*, *requestAttributesEnabled*, *buffered*.
  - org.apache.catalina.valves.RemoteIpValve:
    - if *remoteIpHeader* is defined and isn't set to "x-forwarded-for".
    - if *protocolHeader* is defined and isn't set to "x-forwarded-proto".
    - if you use the following parameters *httpServerPort* and *httpsServerPort* .

Also, please note that Undertow doesn't support JBoss Web **valves**, but some of these may be migrated to Undertow handlers, and JBoss Web subsystem's `migrate` operation do that too.

Here is a list of those valves and their corresponding Undertow handler:

Valve	Handler
org.apache.catalina.valves.AccessLogValve	io.undertow.server.handlers.accesslog.AccessLo
org.apache.catalina.valves.ExtendedAccessLogValve	io.undertow.server.handlers.accesslog.AccessLo
org.apache.catalina.valves.RequestDumperValve	io.undertow.server.handlers.RequestDumpingHa
org.apache.catalina.valves.RewriteValve	io.undertow.server.handlers.SetAttributeHandler
org.apache.catalina.valves.RemoteHostValve	io.undertow.server.handlers.AccessControlListH
org.apache.catalina.valves.RemoteAddrValve	io.undertow.server.handlers.IPAddressAccessCo
org.apache.catalina.valves.RemoteIpValve	io.undertow.server.handlers.ProxyPeerAddressH
org.apache.catalina.valves.StuckThreadDetectionValve	io.undertow.server.handlers.StuckThreadDetecti
org.apache.catalina.valves.CrawlerSessionManagerValve	io.undertow.servlet.handlers.CrawlerSessionMan

The **org.apache.catalina.valves.JDBCAccessLogValve** can't be automatically migrated to **io.undertow.server.handlers.JDBCLogHandler** as the expectations differ.
The migration can be done manually thought :

1. create the driver module and add the driver to the list of available drivers
2. create a datasource pointing to the database where the log entries are going to be stored
3. add an **expression-filter** definition with the following expression:
   "jdbc-access-log(datasource='datasource-jndi-name')

```
<valve name="jdbc" module="org.jboss.as.web"
class-name="org.apache.catalina.valves.JDBCAccessLogValve">
    <param param-name="driverName" param-value="com.mysql.jdbc.Driver" />
    <param param-name="connectionName" param-value="root" />
    <param param-name="connectionPassword" param-value="password" />
    <param param-name="connectionURL"
param-value="jdbc:mysql://localhost:3306/wildfly?zeroDateTimeBehavior=convertToNull" />
    <param param-name="format" param-value="combined" />
</valve>
```

   should become:

```
<subsystem xmlns="urn:jboss:domain:datasources:1.2">
    <datasources>
        <datasource jndi-name="java:jboss/datasources/accessLogDS" pool-name="ccessLogDS"
enabled="true" use-java-context="true">

<connection-url>jdbc:mysql://localhost:3306/wildfly?zeroDateTimeBehavior=convertToNull</conn
<driver>mysql</driver>
            <security>
               <user-name>root</user-name>
               <password>password</password>
            </security>
        </datasource>
...
        <drivers>
            <driver name="mysql" module="com.mysql">
                <driver-class>com.mysql.jdbc.Driver</driver-class>
            </driver>
...
        </drivers>
    </datasources>
</subsystem>
...
<subsystem xmlns="urn:jboss:domain:undertow:3.1"
default-virtual-host="default-virtual-host" default-servlet-container="myContainer"
           default-server="some-server" instance-id="some-id" statistics-enabled="true">
...
    <server name="some-server" default-host="other-host" servlet-container="myContainer">
...
        <host name="other-host" alias="www.mysite.com, ${prop.value:default-alias}"
default-web-module="something.war" disable-console-redirect="true">
            <location name="/" handler="welcome-content" />
            <filter-ref name="jdbc-access"/>
        </host>
    </server>
...
    <filters>
        <expression-filter name="jdbc-access"
expression="jdbc-access-log(datasource='java:jboss/datasources/accessLogDS')" />
...
    </filters>

</subsystem>
```

Please note that any custom valve won't be migrated at all and will just be removed from the configuration. Also the authentication related valves are to be replaced by Undertow authentication mechanisms, and this have to be done manually.

FIXME how this last “manual” replacement is done? Need whole process documented and concrete example

WebSockets

In AS7, to use WebSockets, you had to configure the 'http' **connector** in the **web** subsystem of the server configuration file to use the NIO2 protocol. The following is an example of the Management CLI command to configure WebSockets in the previous releases.

```
/subsystem=web/connector=http/:write-attribute(name=protocol,value=org.apache.coyote.http11.Http11
```

WebSockets are a requirement of the Java EE 7 specification and the default configuration is included in WildFly. More complex WebSocket configuration is done in the **servlet-container** of the **undertow** subsystem of the server configuration file.

You no longer need to configure the server for default WebSocket support.
FIXME isn't <websockets /> required for that?

Messaging Subsystem

WildFly JMS support is provided by ActiveMQ Artemis, instead of HornetQ. It's possible to do a migration of the legacy subsystem configuration, and related persisted data.

Messaging Subsystem Configuration

The extension's module **org.jboss.as.messaging** is replaced by module **org.wildfly.extension.messaging-activemq**, while the subsystem configuration namespace **urn:jboss:domain:messaging:3.0** is replaced by **urn:jboss:domain:messaging-activemq:1.0**.

Management model

In most cases, an effort was made to keep resource and attribute names as similar as possible to those used in previous releases. The following table lists some of the changes.

HornetQ name	ActiveMQ name
hornetq-server	server
hornetq-serverType	serverType
connectors	connector
discovery-group-name	discovery-group

The management operations invoked on the new messaging-subsystem starts with `/subsystem=messaging-activemq/server=X` while the legacy messaging subsystem was at `/subsystem=messaging/hornetq-server=X`.

In case the legacy subsystem configuration is available, such configuration may be migrated to the new subsystem by invoking its `migrate` operation, using the CLI management client:

```
/subsystem=messaging:migrate
```

There is also a `describe-migration` operation that returns a list of all the management operations that are performed to migrate from the legacy subsystem to the new one:

```
/subsystem=messaging:describe-migration
```

Both `migrate` and `describe-migration` will also display a list of migration-warnings if there are some resource or attributes that can not be migrated automatically. The following is a list of these warnings:

- The migrate operation can not be performed: the server must be in admin-only mode
  The `migrate` operation requires starting the server in admin-only mode, which is done by adding parameter `--admin-only` to the server start command, e.g.

  ```
  ./standalone.sh --admin-only
  ```

- Can not migrate attribute local-bind-address from resource X. Use instead the socket-attribute to configure this broadcast-group.
- Can not migrate attribute local-bind-port from resource X. Use instead the socket-binding attribute to configure this broadcast-group.
- Can not migrate attribute group-address from resource X. Use instead the socket-binding attribute to configure this broadcast-group.
- Can not migrate attribute group-port from resource X. Use instead the socket-binding attribute to configure this broadcast-group.
  Broadcast-group resources no longer accept local-bind-address, local-bind-port, group-address, group-port attributes. It only accepts a socket-binding. The warning notifies that resource X has an unsupported attribute. The user will have to set the socket-binding attribute on the resource and ensures it corresponds to a defined socket-binding resource.

- Classes providing the %s are discarded during the migration. To use them in the new messaging-activemq subsystem, you will have to extend the Artemis-based Interceptor.
  Messaging interceptors support is significantly different in WildFly 10, any interceptors configured in the legacy subsystem are discarded during migration. Please refer to the Messaging Interceptors section to learn how to migrate legacy Messaging interceptors.

- Can not migrate the HA configuration of X. Its shared-store and backup attributes holds expressions and it is not possible to determine unambiguously how to create the corresponding ha-policy for the messaging-activemq's server.
  If the hornetq-server X's shared-store or backup attributes hold an expression, such as ${xxx}, then it's not possible to determine the actual ha-policy of the migrated server. In that case, we discard it and the user will have to add the correct ha-policy afterwards (ha-policy is a single resource underneath the messaging-activemq's server resource).

- Can not migrate attribute local-bind-address from resource X. Use instead the socket-binding attribute to configure this discovery-group.Can not migrate attribute local-bind-port from resource X. Use instead the socket-binding attribute to configure this discovery-group.
- Can not migrate attribute group-address from resource X. Use instead the socket-binding attribute to configure this discovery-group.
- Can not migrate attribute group-port from resource X. Use instead the socket-binding attribute to configure this discovery-group.
  discovery-group resources no longer accept local-bind-address, local-bind-port, group-address, group-port attributes. It only accepts a socket-binding. The warning notifies that resource X has an unsupported attribute.
  The user will have to set the socket-binding attribute on the resource and ensures it corresponds to a defined socket-binding resource.

- Can not create a legacy-connection-factory based on connection-factory X. It uses a HornetQ in-vm connector that is not compatible with Artemis in-vm connector
  Legacy subsystem's remote connection-factory resources are migrated into legacy-connection-factory resources, to allow old EAP6 clients to connect to EAP7. However a connection-factory using in-vm will not be migrated, because a in-vm client will be based on EAP7, not EAP 6. In other words, legacy-connection-factory are created only when the CF is using remote connectors, and this warning notifies about in-vm connection-factory X not migrated.

- Can not migrate attribute X from resource Y. The attribute uses an expression that can be resolved differently depending on system properties. After migration, this attribute must be added back with an actual value instead of the expression.
  This warning appears when the migration logic needs to know the concrete value of attribute X during migration, but instead such value includes an expression that's can't be resolved, so the actual value can not be determined, and the attribute is discarded. It happens in several cases, for instance:
    - cluster-connection forward-when-no-consumers. This boolean attribute has been replaced by the message-load-balancing-type attribute (which is an enum of OFF, STRICT, ON_DEMAND)
    - broadcast-group and discovery-group's jgroups-stack and jgroups-channel attributes. They reference other resources and we no longer accept expressions for them.

- Can not migrate attribute X from resource Y. This attribute is not supported by the new messaging-activemq subsystem.
  Some attributes are no longer supported in the new messaging-activemq subsystem and are simply discarded:
    - hornetq-server's failback-delay
    - http-connector's use-nio attribute
    - http-acceptor's use-nio attribute
    - remote-connector's use-nio attribute
    - remote-acceptor's use-nio attribute

XML Configuration

The XML configuration has changed significantly with the new messaging-activemq subsystem to provide a XML scheme more consistent with other WildFly subsystems.
It is not advised to change the XML configuration of the legacy messaging subsystem to conform to the new messaging-activemq subsystem. Instead, invoke the legacy subsystem `migrate` operation. This operation will write the XML configuration of the new **messaging-activemq** subsystem as a part of its execution.

Messaging Interceptors

Messaging Interceptors are significantly different in EAP 7, requiring both code and configuration changes by the user. In concrete the interceptor base Java class is now **org.apache.artemis.activemq.api.core.interceptor.Interceptor**, and the user interceptor implementation classes may now be loaded by any server module. Note that prior to EAP 7 the interceptor classes could only be installed by adding these to the HornetQ module, thus requiring the user to change such module XML descriptor, its **module.xml**.
With respect to the server XML configuration, the user must now specify the module to load its interceptors in the new **messaging-activemq** subsystem XML config, e.g:

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:1.0">
    <server name=“default”>
       ...
        <incoming-interceptors>
            <class name="org.foo.incoming.myInterceptor" module="org.foo" />
            <class name="org.bar.incoming.myOtherInterceptor" module="org.bar" />
        </incoming-interceptors>
        <outgoing-interceptors>
            <class name="org.foo.outgoing.myInterceptor" module="org.foo" />
            <class name="org.bar.outgoing.myOtherInterceptor" module="org.bar" />
        </outgoing-interceptors>
   </server>
</subsystem>
```

JMS Destinations

In previous releases, JMS destination queues were configured in the <jms-destinations> element under the hornetq-server section of the **messaging** subsystem.

```
<jms-destinations>
<jms-queue name="testQueue">
<entry name="queue/test"/>
<entry name="java:jboss/exported/jms/queue/test"/>
</jms-queue>
</jms-destinations>
```

In WildFly, the JMS destination queue is configured in the default server of the messaging-activemq subsystem.

```
<jms-queue name="testQueue" entries="queue/test java:jboss/exported/jms/queue/test"/>
```

Messaging Logging

The prefix of messaging log messages in WildFly is **WFLYMSGAMQ**, instead of **WFLYMSG**.

Messaging Data

The location of the messaging data has been changed in the new messaging-activemq subsystem:

- messagingbindings/ -> activemq/bindings/
- messagingjournal/ -> activemq/journal/
- messaginglargemessages/ -> activemq/largemessages/
- messagingpaging/ -> activemq/paging/

To migrate legacy messaging data, you will have to export the directories used by the legacy messaging subsystem and import them into the new subsystem's server by using its `import-journal operation:`

```
/subsystem=messaging-activemq/server=default:import-journal(file=<path to XML dump>)
```

The XML dump is a XML file generated by HornetQ `XmlDataExporter` util class.

6.24.4 Application Migration

Before you migrate your application, you should be aware that some features that were available in previous releases are now deprecated or missing.

EJBs

CMP Entity EJBs

Container-Managed Persistence entity beans support is optional in Java EE 7, and WildFly does not provide support for these.

CMP entity beans are defined in the **ejb-jar.xml** descriptor, in concrete an entity bean is CMP only if the **<entity/>**'s child element named **persistence-type** is included and has a value of **Container**. An example:

```
<?xml version="1.1" encoding="UTF-8"?>
<ejb-jar xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd"
         version="3.1">
    <enterprise-beans>
        <entity>
            <ejb-name>SimpleBMP</ejb-name>
            <local-home>org.jboss.as.test.integration.ejb.entity.bmp.BMPLocalHome</local-home>
            <local>org.jboss.as.test.integration.ejb.entity.bmp.BMPLocalInterface</local>
            <ejb-class>org.jboss.as.test.integration.ejb.entity.bmp.SimpleBMPBean</ejb-class>
            <persistence-type>Container</persistence-type>
            <prim-key-class>java.lang.Integer</prim-key-class>
            <reentrant>true</reentrant>
        </entity>
    </enterprise-beans>
</ejb-jar>
```

CMP entity beans should be replaced by JPA entities.

EJB Client

Default Remote Connection Port

The default remote connection port has changed from '4447' to '8080'.

In JBoss AS7, the **jboss-ejb-client.properties** file looked similar to the following:

```
remote.connectionprovider.create.options.org.xnio.Options.SSL_ENABLED=false
remote.connections=default
remote.connection.default.host=localhost
remote.connection.default.port=4447
remote.connection.default.connect.options.org.xnio.Options.SASL_POLICY_NOANONYMOUS=false
```

In WildFly, the properties file looks like this:

```
remote.connectionprovider.create.options.org.xnio.Options.SSL_ENABLED=false
remote.connections=default
remote.connection.default.host=localhost
remote.connection.default.port=8080
remote.connection.default.connect.options.org.xnio.Options.SASL_POLICY_NOANONYMOUS=false
```

Default Connector

In WildFly, the default connector has changed from "remoting" to "http-remoting". This change impacts clients that use libraries from one release of JBoss and to connect to server in a different release.

- If a client application uses the EJB client library from JBoss AS 7 and wants to connect to WildFly 10 server, the server must be configured to expose a remoting connector on a port other than "8080". The client must then connect using that newly configured connector.
- A client application that uses the EJB client library from WildFly 10 and wants to connect to a JBoss AS 7 server must be aware that the server instance does not use the http-remoting connector and instead uses a remoting connector. This is achieved by defining a new client-side connection property.

  ```
  remote.connection.default.protocol=remote
  ```

External applications using JNDI, to remotely lookup up EJBs in a WildFly 10 server, may also need to be migrated, please refer to Remote JNDI Clients section for further information.

JMS

Proprietary JMS Resource Definitions

The proprietary XML descriptors, previously used to setup JMS resources, are deprecated in WildFly. Java EE 7 (section EE.5.18) standardised such functionality.

The deprecated descriptors are files bundled in the application package, which name ends with **-jms.xml**. Their namespace has been changed to **urn:jboss:messaging-activemq-deployment:1.0**.

External JMS Clients

JMS Resources are remotely looked up using JNDI, and looking up resources in a WildFly 10 server may require changes in the application code, please refer to Remote JNDI Clients section for further information.

JPA (and Hibernate)

Applications That Plan to Use Hibernate ORM 5.0

WildFly ships with Hibernate ORM 5.0 and those libraries are implicitly added to the application classpath when a persistence.xml is detected during deployment. If your application uses JPA, it will default to using the Hibernate ORM 5.0 libraries.

Hibernate ORM 5.0 introduces:

- Redesigned metamodel - Complete replacement for the current org.hibernate.mapping code
- Query parser - Improved query parser based on Antlr 3/4
- Multi-tenancy improvements - Discriminator-based multi-tenancy
- Follow-on fetches - Two-phase loading via LoadPlans/EntityGraphs

Applications that currently use Hibernate ORM 4.0 - 4.3

If your application needs second-level cache enabled, you should migrate to Hibernate ORM 5.0, which is integrated with Infinispan 8.0. Applications written with Hibernate ORM 4.x can still use Hibernate 4.x if you define a custom JBoss module with Hibernate 4.x JARs and exclude the Hibernate 5 classes from your application. It is recommended that you migrate your application to use Hibernate 5.

For information about the changes implemented between Hibernate 4 and Hibernate 5, see https://github.com/hibernate/hibernate-orm/blob/master/migration-guide.adoc

Applications that currently use Hibernate 3

The integration classes that made it easier to use Hibernate 3 in AS 7 were removed from WildFly 10. If your application still uses Hibernate 3 libraries, it is strongly recommended that you migrate your application to use Hibernate 5 as Hibernate 3 will no longer work in WildFly without a lot of effort. If you can not migrate to Hibernate 5, you must define a custom JBoss Module for the Hibernate 3 classes and exclude the Hibernate 5 classes from your application.

Web Applications

JBoss Web Valves

Undertow does not support the JBoss Web Valve functionality. This can be replaced by Undertow Handlers (see http://undertow.io/undertow-docs/undertow-docs-1.3.0/index.html#undertow-handler-authors-guide for more).

List of valves that were provided with JBoss Web, together with a corresponding Undertow handler, is provided above, in the section on the JBoss Web subsystem.

JBoss Web Valves are specified in the proprietary **jboss-web.xml** descriptor, through **<valve />** element(s). These can be replaced using the **<http-handler />** element(s). For example:

```
<jboss-web>
    <valve>
        <class-name>org.apache.catalina.valves.RequestDumperValve</class-name>
        <module>org.jboss.as.web</module>
    </valve>
</jboss-web>
```

can be replaced by

```
<jboss-web>
    <http-handler>
        <class-name>io.undertow.server.handlers.RequestDumpingHandler</class-name>
        <module>io.undertow.core</module>
    </http-handler>
</jboss-web>
```

Web Services

CXF Spring Webservices

The setup of web service's endpoints and clients, through a Spring XML descriptor, driving a CXF bus creation, is no longer supported in WildFly.

Any application containing a jbossws-cxf.xml must migrate all functionality specified in such XML descriptor, mostly already supported by the JAX-WS specification, included in Java EE 7. It is still possible to rely on direct Apache CXF API usage, loosing the Java EE portability of the application, for instance when specific Apache CXF functionalities are needed. Please refer to the Apache CXF Integration document for further information.

RPC

JAX-RPC is an API for building Web services and clients that used remote procedure calls (RPC) and XML, which was deprecated in Java EE 6, and is no longer supported by WildFly.

JAX-RPC Web Services may be identified by the presence of the XML descriptor named web services.xml, containing a **<webservice-description/>** element that includes a child element named **<jaxrpc-mapping-file/>**. An example:

```
<webservices xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://www.ibm.com/webservices/xsd/j2ee_web_services_1_1.xsd" version="1.1">
    <webservice-description>
        <webservice-description-name>HelloService</webservice-description-name>
        <wsdl-file>WEB-INF/wsdl/HelloService.wsdl</wsdl-file>
        <jaxrpc-mapping-file>WEB-INF/mapping.xml</jaxrpc-mapping-file>
        <port-component>
            <port-component-name>Hello</port-component-name>
            <wsdl-port>HelloPort</wsdl-port>

<service-endpoint-interface>org.jboss.chap12.hello.Hello</service-endpoint-interface>
            <service-impl-bean>
                <servlet-link>HelloWorldServlet</servlet-link>
            </service-impl-bean>
        </port-component>
    </webservice-description>
</webservices>
```

Applications using JAX-RPC should be migrated to use JAX-WS, the current Java EE standard web service framework.

RS 2.0

JSR 339: JAX-RS 2.0: The Java API for RESTful Web Services specification is located here: https://jcp.org/en/jsr/detail?id=339

Some changes to the `MessageBodyWriter` interface may represent a backward incompatible change with respect to JAX-RS 1.X.

Be sure to define an @Produces or @Consumes for your endpoints. Failure to do so may result in an error similar to the following.

```
org.jboss.resteasy.core.NoMessageBodyWriterFoundFailure: Could not find MessageBodyWriter for
response object of type: <OBJECT> of media type: <CONTENT_TYPE>
```

REST Client API

Some REST Client API classes and methods are deprecated, for example: org.jboss.resteasy.client.ClientRequest and org.jboss.resteasy.client.ClientResponse. Instead, use org.jboss.resteasy.client.jaxrs.ResteasyClient and javax.ws.rs.core.Response. See the `resteasy-jaxrs-client quickstart` for an example of an external JAX-RS RestEasy client that interacts with a JAX-RS Web service.

Application Clustering

HA Singleton

JBoss AS7 introduced singleton services - a mechanism for installing an service such that it would only start on one node in the cluster at a time, a HA Singleton. Such mechanism required usage of a private JBoss EAP Clustering API, designed around the class **org.jboss.as.clustering.singleton.SingletonService**, and was documented in detail at https://access.redhat.com/documentation/en-US/JBoss_Enterprise_Application_Platform/6.4/html/Developme , and while not difficult to implement, the installation process suffered from a couple shortcomings:

- Installing multiple singleton services within a single deployment caused the deployer to hang.
- Installing a singleton service required the user to specify several private module dependencies in /META-INF/MANIFEST.MF

WildFly 10 introduces a new public API for building such services, which significantly simplifies the process, and solves the issues found in the legacy solution. The JBoss EAP 7 Quickstart application named **cluster-ha-singleton** examples a HA Singleton implementation using the new API, and may be found at https://github.com/jboss-developer/jboss-eap-quickstarts/tree/7.0.x-develop/cluster-ha-singleton .
FIXME: community URLs instead

Stateful Session EJB Clustering

WildFly 10 no longer requires Stateful Session EJBs to use the **org.jboss.ejb3.annotation.Clustered** annotation to enable clustering behaviour. By default, if the server is started using an HA profile, the state of your SFSBs will be replicated automatically. Disabling this behaviour is achievable on a per-EJB basis, by annotating your bean using **@Stateful(passivationCapable=false)**, which is new to the EJB 3.2 specification; or globally through the configuration of the EJB3 subsystem, in the server configuration.

Note that the **@Clustered** annotation, if used by an application, is simply ignored, the application deployment will not fail.

Web Session Clustering

WildFly 10 introduces a new web session clustering implementation, replacing the one found in AS7, which has been around for ages (since JBoss AS 3.2!), and was tightly coupled to the legacy JBoss Web subsystem source code. The most relevant changes in the new implementation are:

- Introduction of a proper session manager SPI, and an Infinispan implementation of it, decoupled from the web subsystem implementation
- Sessions are implemented as a facade over one or more cache entries, which means that the container's session manager itself does not retain a separate reference to each HttpSession
- Pessimistic locking of cache entries effectively ensures that only a single client on a single node ever accesses a given session at any given time
- Usage of cache entry grouping, instead of atomic maps, to ensure that multiple cache entries belonging to the same session are co-located.
- Session operations within a request only ever use a single batch/transaction. This results in fewer RPCs per request.
- Support for write-through cache stores, as well as passivation-only cache stores.

With respect to applications, the new web session clustering implementation deprecates/reinterprets much of the related configuration, which is included in JBoss's proprietary web application XML descriptor, **jboss-web.xml**:

- **<max-active-sessions/>**
  Previously, session creation would fail if an additional session would cause the number of active sessions to exceed the value specified by <**max-active-sessions/>**.
  In the new implementation, **<max-active-sessions/>** is used to enable session passivation. If session creation would cause the number of active sessions to exceed **<max-active-sessions/>**, then the oldest session known to the session manager will passivate to make room for the new session.

- **<passivation-config/>**
  This configuration element and its sub-elements are no longer used in WildFly.

- **<use-session-passivation/>**
  Previously, passivation was enabled via this attribute, yet in the new implementation, passivation is enabled by specifying a non-negative value for **<max-active-sessions/>**.

- **<passivation-min-idle-time/>**
  Previously, sessions needed to be active for at least a specific amount of time before becoming a candidate for passivation. This could cause session creation to fail, even when passivation was enabled.
  The new implementation does not support this logic and thus avoids this DoS vulnerability.

- **<passivation-max-idle-time/>**
  Previously, a session would be passivated after it was idle for a specific amount of time.
  The new implementation does not support eager passivation - only lazy passivation. Sessions are only passivated when necessary to comply with **<max-active-sessions/>**.

- **<replication-config/>**
  The new implementation deprecates a number of sub-elements:

- **<replication-trigger/>**
  Previously, session attributes could be treated as either mutable or immutable depending on the values specified by **<replication-trigger/>**:
  - SET treated all attributes as immutable, requiring a separate HttpSession.setAttribute(...) to indicate that the value changed.
  - SET_AND_GET treated all session attributes as mutable.
  - SET_AND_NON_PRIMITIVE_GET recognised a small set of types (i.e. strings and boxed primitives) as immutable, and assumed that any other attribute was mutable.
    The new implementation replaces this configuration option with a single, robust strategy.
    Session attributes are assumed to be mutable unless one of the following is true:
  - The value is a known immutable value:
    - null
    - java.util.Collections.EMPTY_LIST, EMPTY_MAP, EMPTY_SET
  - The value type is or implements a known immutable type:
    - Boolean, Byte, Character, Double, Float, Integer, Long, Short
    - java.lang.Enum, StackTraceElement, String
    - java.io.File, java.nio.file.Path
    - java.math.BigDecimal, BigInteger, MathContext
    - java.net.InetAddress, InetSocketAddress, URI, URL
    - java.security.Permission
    - java.util.Currency, Locale, TimeZone, UUID
  - The value type is annotated with @org.wildfly.clustering.web.annotation.Immutable

- **<use-jk/>**
  Previously, the instance-id of the node handling a given request was appended to the jsessionid (for use by load balancers such as mod_jk, mod_proxy_balancer, mod_cluster, etc.) depending on the value specified for **<use-jk/>**. In the new implementation, the instance-id, if defined, is always appended to the jsessionid.

- **<max-unreplicated-interval/>**
  Previously, this configuration option was an optimization that would prevent the replicate of a session’s timestamp if no session attribute was changed. While this sounds nice, in practice it doesn't prevent any RPCs, since session access requires cache transaction RPCs regardless of whether any session attributes changed. In the new implementation, the timestamp of a session is replicated on every request. This prevents stale session meta data following failover.

- **<snapshot-mode/>**
  Previously, one could configure **<snapshot-mode/>** as INSTANT or INTERVAL. Infinispan’s replication queue renders this configuration option obsolete.

- **<snapshot-interval/>**
  Only relevant for **<snapshot-mode>INTERVAL</snapshot-mode>**. See above.

- **<session-notification-policy/>**
  Previously, the value defined by this attribute defined a policy for triggering session events. In the new implementation, this behaviour is spec-driven and not configurable.

Other Specifications and Frameworks

Remote JNDI Clients

WildFly 10's default JNDI Provider URL has changed, which means that external applications, using JNDI to lookup remote resources, for instance an EJB or a JMS Queue, may need to change the value for the JNDI **InitialContext** environment's property named **java.naming.provider.url**. The default URL scheme is now **http-remoting**, and the default URL port is now **8080**.

As an example, considering the application server host is **localhost**, then clients previously accessing JBoss AS7 would use

```
java.naming.factory.initial=org.jboss.naming.remote.client.InitialContextFactory
java.naming.provider.url=remote://localhost:4447
```

while clients now accessing WildFly should use instead

```
java.naming.factory.initial=org.jboss.naming.remote.client.InitialContextFactory
java.naming.provider.url=http-remoting://localhost:8080
```

88

The specification which aimed to standardise deployment tasks got very little adoption, due to much more "feature rich" proprietary solutions already included in every vendor application server. It was no surprise that JSR-88 support was dropped from Java EE 7, and WildFly followed that and dropped support too.

A JSR-88 deployment plan is identified by a XML descriptor named **deployment-plan.xml**, bundled in a zip/jar archive.

Module Dependencies

Applications defining dependencies to WildFly modules, through the application's package MANIFEST.MF or jboss-deployment-structure.xml, may be referencing missing modules. When migrating an application, relying on such functionality, the presence of the referenced modules should be validated in advance.

6.25 How do I migrate my application to WildFly from other application servers

6.25.1 Choose from the list below:

- How do I migrate my application from WebLogic to WildFly
- How do I migrate my application from WebSphere to WildFly

6.25.2 How do I migrate my application from WebLogic to WildFly

The purpose of this guide is to document the application changes that are needed to successfully run and deploy WebLogic applications on WildFly.

Feel free to add content in any way you prefer. You do not need to follow the template below. This is a work in progress.

Introduction

About this Guide

The purpose of this document is to guide you through the planning process and migration of fairly simple and standard Oracle WebLogic applications to WildFly. O

6.25.3 How do I migrate my application from WebSphere to WildFly

The purpose of this guide is to document the application changes that are needed to successfully run and deploy WebLogic applications on WildFly.

Feel free to add content in any way you prefer. You do not need to follow the template below. This is a work in progress.

Introduction

About this Guide

The purpose of this document is to guide you through the planning process and migration of fairly simple and standard Oracle WebLogic applications to WildFly.

6.26 Implicit module dependencies for deployments

As explained in the Class Loading in WildFly article, WildFly 8 is based on module classloading. A class within a module B isn't visible to a class within a module A, unless module B adds a dependency on module A. Module dependencies can be explicitly (as explained in that classloading article) or can be "implicit". This article will explain what implicit module dependencies mean and how, when and which modules are added as implicit dependencies.

6.26.1 What's an implicit module dependency?

Consider an application deployment which contains EJBs. EJBs typically need access to classes from the javax.ejb.* package and other Java EE API packages. The jars containing these packages are already shipped in WildFly and are available as "modules". The module which contains the javax.ejb.* classes has a specific name and so does the module which contains all the Java EE API classes. For an application to be able to use these classes, it has to add a dependency on the relevant modules. Forcing the application developers to add module dependencies like these (i.e. dependencies which can be "inferred") isn't a productive approach. Hence, whenever an application is being deployed, the deployers within the server, which are processing this deployment "implicitly" add these module dependencies to the deployment so that these classes are visible to the deployment at runtime. This way the application developer doesn't have to worry about adding them explicitly. How and when these implicit dependencies are added is explained in the next section.

6.26.2 How and when is an implicit module dependency added?

When a deployment is being processed by the server, it goes through a chain of "deployment processors". Each of these processors will have a way to check if the deployment meets a certain criteria and if it does, the deployment processor adds a implicit module dependency to that deployment. Let's take an example - Consider (again) an EJB3 deployment which has the following class:

MySuperDuperBean.java

```
@Stateless
public class MySuperDuperBean {

...

}
```

As can be seen, we have a simple @Stateless EJB. When the deployment containing this class is being processed, the EJB deployment processor will see that the deployment contains a class with the @Stateless annotation and thus identifies this as a EJB deployment. **This is just one of the several ways, various deployment processors can identify a deployment of some specific type.** The EJB deployment processor will then add an implicit dependency on the Java EE API module, so that all the Java EE API classes are visible to the deployment.

Some subsystems will always add a API classes, even if the trigger condition is not met. These are listed separately below.

In the next section, we'll list down the implicit module dependencies that are added to a deployment, by various deployers within WildFly.

6.26.3 Which are the implicit module dependencies?

Subsystem responsible for adding the implicit dependency	Dependencies that are always added	Dependencies that are added if a trigger condition is met
Core Server	• javax.api • sun.jdk • org.jboss.vfs	

Batch Subsystem	• javax.batch.api	
EE Subsystem	• javaee.api	
EJB3 subsystem		• javaee.api
JAX-RS (Resteasy) subsystem	• javax.xml.bind.api	• org.jboss.resteasy.resteasy-atom-provider • org.jboss.resteasy.resteasy-cdi • org.jboss.resteasy.resteasy-jaxrs • org.jboss.resteasy.resteasy-jaxb-provider • org.jboss.resteasy.resteasy-jackson-provider • org.jboss.resteasy.resteasy-jsapi • org.jboss.resteasy.resteasy-multipart-provide • org.jboss.resteasy.async-http-servlet-30
JCA subsystem	• javax.resource.api	• javax.jms.api • javax.validation.api • org.jboss.logging • org.jboss.ironjacamar.api • org.jboss.ironjacamar.impl • org.hibernate.validator
JPA (Hibernate) subsystem	• javax.persistence.api	• javaee.api • org.jboss.as.jpa • org.hibernate

Logging Subsystem	• org.jboss.logging • org.apache.commons.logging • org.apache.log4j • org.slf4j • org.jboss.logging.jul-to-slf4j-stub	
SAR Subsystem		• org.jboss.logging • org.jboss.modules
Security Subsystem	• org.picketbox	
Web Subsystem		• javaee.api • com.sun.jsf-impl • org.hibernate.validator • org.jboss.as.web • org.jboss.logging
Web Services Subsystem	• org.jboss.ws.api • org.jboss.ws.spi	
Weld (CDI) Subsystem		• javax.persistence.api • javaee.api • org.javassist • org.jboss.interceptor • org.jboss.as.weld • org.jboss.logging • org.jboss.weld.core • org.jboss.weld.api • org.jboss.weld.spi

6.27 RS Reference Guide

This page outlines the three options you have for deploying JAX-RS applications in WildFly 8. These three methods are specified in the JAX-RS 1.1 specification in section 2.3.2.

6.27.1 Subclassing javax.ws.rs.core.Application and using @ApplicationPath

This is the easiest way and does not require any xml configuration. Simply include a subclass of `javax.ws.rs.core.Application` in your application, and annotate it with the path that you want your JAX-RS classes to be available. For example:

```
@ApplicationPath("/mypath")
public class MyApplication extends Application {
}
```

This will make your JAX-RS resources available under `/mywebappcontext/mypath`.

Note that that the path is `/mypath` not `/mypath/*`

6.27.2 Subclassing javax.ws.rs.core.Application and using web.xml

If you do not wish to use `@ApplicationPath` but still need to subclass `Application` you can set up the JAX-RS mapping in web.xml:

```
public class MyApplication extends Application {
}
```

```
<servlet-mapping>
   <servlet-name>com.acme.MyApplication</servlet-name>
   <url-pattern>/hello/*</url-pattern>
</servlet-mapping>
```

This will make your JAX-RS resources available under `/mywebappcontext/hello`.

You can also use this approach to override an application path set with the `@ApplicationPath` annotation.

6.27.3 Using web.xml

If you don't wan't to subclass `Application` you can set the JAX-RS mapping in web.xml as follows:

```
<servlet-mapping>
   <servlet-name>javax.ws.rs.core.Application</servlet-name>
   <url-pattern>/hello/*</url-pattern>
</servlet-mapping>
```

This will make your JAX-RS resources available under `/mywebappcontext/hello`.

Note that you only have to add the mapping, not the corresponding servlet. The server is responsible for adding the corresponding servlet automatically.

6.28 JNDI Reference

6.28.1 Overview

WildFly offers several mechanisms to retrieve components by name. Every WildFly instance has it's own local JNDI namespace (`java:`) which is unique per JVM. The layout of this namespace is primarily governed by the Java EE specification. Applications which share the same WildFly instance can use this namespace to intercommunicate. In addition to local JNDI, a variety of mechanisms exist to access remote components.

- Client JNDI - This is a mechanism by which remote components can be accessed using the JNDI APIs, but ***without network round-trips***. This approach is the most efficient, and ***removes a potential single point of failure***. For this reason, it is highly recommended to use Client JNDI over traditional remote JNDI access. However, to make this possible, it does require that all names follow a strict layout, so user customizations are not possible. Currently only access to remote EJBs is supported via the `ejb:` namespace. Future revisions will likely add a JMS client JNDI namespace.
- Traditional Remote JNDI - This is a more familiar approach to EE application developers, where the client performs a remote component name lookup against a server, and a proxy/stub to the component is serialized as part of the name lookup and returned to the client. The client then invokes a method on the proxy which results in another remote network call to the underlying service. In a nutshell, traditional remote JNDI involves two calls to invoke an EE component, whereas Client JNDI requires one. It does however allow for customized names, and for a centralised directory for multiple application servers. This centralized directory is, however, *a single point of failure*.
- EE Application Client / Server-To-Server Delegation - This approach is where local names are bound as an *alias* to a remote name using one of the above mechanisms. This is useful in that it allows applications to only ever reference standard portable Java EE names in both code and deployment descriptors. It also allows for the application to be unaware of network topology details/ This can even work with Java SE clients by using the little known EE Application Client feature. This feature allows you to run an extremely minimal AS server around your application, so that you can take advantage of certain core services such as naming and injection.

6.28.2 Local JNDI

The Java EE platform specification defines the following JNDI contexts:

- `java:comp` - The namespace is scoped to the current component (i.e. EJB)
- `java:module` - Scoped to the current module
- `java:app` - Scoped to the current application
- `java:global` - Scoped to the application server

In addition to the standard namespaces, WildFly also provides the following two global namespaces:

- java:jboss
- java:/

Only entries within the `java:jboss/exported` context are accessible over remote JNDI.

For web deployments `java:comp` is aliased to `java:module`, so EJB's deployed in a war do not have their own comp namespace.

Binding entries to JNDI

There are several methods that can be used to bind entries into JNDI in WildFly.

Using a deployment descriptor

For Java EE applications the recommended way is to use a deployment descriptor to create the binding. For example the following `web.xml` binds the string `"Hello World"` to `java:global/mystring` and the string `"Hello Module"` to `java:comp/env/hello` (any non absolute JNDI name is relative to `java:comp/env` context).

```
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
         version="3.1">
    <env-entry>
        <env-entry-name>java:global/mystring</env-entry-name>
        <env-entry-type>java.lang.String</env-entry-type>
        <env-entry-value>Hello World</env-entry-value>
    </env-entry>
    <env-entry>
        <env-entry-name>hello</env-entry-name>
        <env-entry-type>java.lang.String</env-entry-type>
        <env-entry-value>Hello Module</env-entry-value>
    </env-entry>
</web-app>
```

For more details, see the Java EE Platform Specification.

Programatically

Java EE Applications

Standard Java EE applications may use the standard JNDI API, included with Java SE, to bind entries in the global namespaces (the standard `java:comp`, `java:module` and `java:app` namespaces are read-only, as mandated by the Java EE Platform Specification).

```
InitialContext initialContext = new InitialContext();
  initialContext.bind("java:global/a", 100);
```

There is no need to unbind entries created programatically, since WildFly tracks which bindings belong to a deployment, and the bindings are automatically removed when the deployment is undeployed.

WildFly Modules and Extensions

With respect to code in WildFly Modules/Extensions, which is executed out of a Java EE application context, using the standard JNDI API may result in a UnsupportedOperationException if the target namespace uses a WritableServiceBasedNamingStore. To work around that, the bind() invocation needs to be wrapped using WildFly proprietary APIs:

```
InitialContext initialContext = new InitialContext();
  WritableServiceBasedNamingStore.pushOwner(serviceTarget);
  try {
    initialContext.bind("java:global/a", 100);
  } finally {
    WritableServiceBasedNamingStore.popOwner();
  }
```

The ServiceTarget removes the bind when uninstalled, thus using one out of the module/extension domain usage should be avoided, unless entries are removed using unbind().

Naming Subsystem Configuration

It is also possible to bind to one of the three global namespaces using configuration in the naming subsystem. This can be done by either editing the `standalone.xml/domain.xml` file directly, or through the management API.

Four different types of bindings are supported:

- Simple - A primitive or java.net.URL entry (default is `java.lang.String`).
- Object Factory - This allows to to specify the `javax.naming.spi.ObjectFactory` that is used to create the looked up value.
- External Context - An external context to federate, such as an LDAP Directory Service
- Lookup - The allows to create JNDI aliases, when this entry is looked up it will lookup the target and return the result.

An example standalone.xml might look like:

```
<subsystem xmlns="urn:jboss:domain:naming:2.0" >
  <bindings>
    <simple name="java:global/a" value="100" type="int" />
    <simple name="java:global/jbossDocs" value="https://docs.jboss.org" type="java.net.URL" />
    <object-factory name="java:global/b" module="com.acme" class="org.acme.MyObjectFactory" />
    <external-context name="java:global/federation/ldap/example”
class="javax.naming.directory.InitialDirContext" cache="true">
      <environment>
        <property name="java.naming.factory.initial" value=“com.sun.jndi.ldap.LdapCtxFactory” />
        <property name="java.naming.provider.url" value=“ldap://ldap.example.com:389” />
        <property name="java.naming.security.authentication" value=“simple” />
        <property name="java.naming.security.principal" value=“uid=admin,ou=system” />
        <property name="java.naming.security.credentials" value=“secret” />
      </environment>
    </external-context>
    <lookup name="java:global/c" lookup="java:global/b" />
 </bindings>
</subsystem>
```

The CLI may also be used to bind an entry. As an example:

```
/subsystem=naming/binding=java\:global\/mybinding:add(binding-type=simple, type=long,
value=1000)
```

WildFly's Administrator Guide includes a section describing in detail the Naming subsystem configuration.

Retrieving entries from JNDI

Resource Injection

For Java EE applications the recommended way to lookup a JNDI entry is to use `@Resource` injection:

```
@Resource(lookup = "java:global/mystring")
  private String myString;

  @Resource(name = "hello")
  private String hello;

  @Resource
  ManagedExecutorService executor;
```

Note that `@Resource` is more than a JNDI lookup, it also binds an entry in the component's JNDI environment. The new bind JNDI name is defined by `@Resource`'s `name` attribute, which value, if unspecified, is the Java type concatenated with `/` and the field's name, for instance `java.lang.String/myString`. More, similar to when using deployment descriptors to bind JNDI entries. unless the name is an absolute JNDI name, it is considered relative to `java:comp/env`. For instance, with respect to the field named `myString` above, the `@Resource`'s `lookup` attribute instructs WildFly to lookup the value in `java:global/mystring`, bind it in `java:comp/env/java.lang.String/myString`, and then inject such value into the field.

With respect to the field named `hello`, there is no `lookup` attribute value defined, so the responsibility to provide the entry's value is delegated to the deployment descriptor. Considering that the deployment descriptor was the `web.xml` previously shown, which defines an environment entry with same `hello` name, then WildFly inject the valued defined in the deployment descriptor into the field.

The `executor` field has no attributes specified, so the bind's name would default to `java:comp/env/javax.enterprise.concurrent.ManagedExecutorService/executor`, but there is no such entry in the deployment descriptor, and when that happens it's up to WildFly to provide a default value or null, depending on the field's Java type. In this particular case WildFly would inject the default instance of a managed executor service, the value in `java:comp/DefaultManagedExecutorService`, as mandated by the EE Concurrency Utilities 1.0 Specification (JSR 236).

Standard Java SE JNDI API

Java EE applications may use, without any additional configuration needed, the standard JNDI API to lookup an entry from JNDI:

```
String myString = (String) new InitialContext().lookup("java:global/mystring");
```

or simply

```
String myString = InitialContext.doLookup("java:global/mystring");
```

6.28.3 Remote JNDI

WildFly supports two different types of remote JNDI. The old jnp based JNDI implementation used in JBoss AS versions prior to 7.x is no longer supported.

remote:

The `remote:` protocol uses the WildFly remoting protocol to lookup items from the servers local JNDI. To use it, you must have the appropriate jars on the class path, if you are maven user can be done simply by adding the following to your `pom.xml`:

```
<dependency>
  <groupId>org.wildfly</groupId>
  <artifactId>wildfly-ejb-client-bom</artifactId>
  <version>8.0.0.Final</version>
  <type>pom</type>
  <scope>compile</scope>
</dependency>
```

If you are not using maven a shaded jar that contains all required classes
can be found in the `bin/client` directory of WildFly's distribution.

```
final Properties env = new Properties();
env.put(Context.INITIAL_CONTEXT_FACTORY,
org.jboss.naming.remote.client.InitialContextFactory.class.getName());
env.put(Context.PROVIDER_URL, "remote://localhost:4447");
remoteContext = new InitialContext(env);
```

ejb:

The ejb: namespace is provided by the jboss-ejb-client library. This protocol allows you to look up EJB's, using their application name, module name, ejb name and interface type.

This is a client side JNDI implementation. Instead of looking up an EJB on the server the lookup name contains enough information for the client side library to generate a proxy with the EJB information. When you invoke a method on this proxy it will use the current EJB client context to perform the invocation. If the current context does not have a connection to a server with the specified EJB deployed then an error will occur. Using this protocol it is possible to look up EJB's that do not actually exist, and no error will be thrown until the proxy is actually used. The exception to this is stateful session beans, which need to connect to a server when they are created in order to create the session bean instance on the server.

Some examples are:

```
ejb:myapp/myejbjar/MyEjbName!com.test.MyRemoteInterface
ejb:myapp/myejbjar/MyStatefulName!comp.test.MyStatefulRemoteInterface?stateful
```

The first example is a lookup of a singleton, stateless or EJB 2.x home interface. This lookup will not hit the server, instead a proxy will be generated for the remote interface specified in the name. The second example is for a stateful session bean, in this case the JNDI lookup will hit the server, in order to tell the server to create the SFSB session.

For more details on how the server connections are configured, please see EJB invocations from a remote client using JNDI.

6.28.4 Local JNDI

The Java EE platform specification defines the following JNDI contexts:

- `java:comp` - The namespace is scoped to the current component (i.e. EJB)
- `java:module` - Scoped to the current module
- `java:app` - Scoped to the current application
- `java:global` - Scoped to the application server

In addition to the standard namespaces, WildFly also provides the following two global namespaces:

- java:jboss
- java:/

Only entries within the `java:jboss/exported` context are accessible over remote JNDI.

For web deployments `java:comp` is aliased to `java:module`, so EJB's deployed in a war do not have their own comp namespace.

Binding entries to JNDI

There are several methods that can be used to bind entries into JNDI in WildFly.

Using a deployment descriptor

For Java EE applications the recommended way is to use a deployment descriptor to create the binding. For example the following `web.xml` binds the string `"Hello World"` to `java:global/mystring` and the string `"Hello Module"` to `java:comp/env/hello` (any non absolute JNDI name is relative to `java:comp/env` context).

```
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
         version="3.1">
    <env-entry>
        <env-entry-name>java:global/mystring</env-entry-name>
        <env-entry-type>java.lang.String</env-entry-type>
        <env-entry-value>Hello World</env-entry-value>
    </env-entry>
    <env-entry>
        <env-entry-name>hello</env-entry-name>
        <env-entry-type>java.lang.String</env-entry-type>
        <env-entry-value>Hello Module</env-entry-value>
    </env-entry>
</web-app>
```

For more details, see the Java EE Platform Specification.

Programatically

Java EE Applications

Standard Java EE applications may use the standard JNDI API, included with Java SE, to bind entries in the global namespaces (the standard `java:comp`, `java:module` and `java:app` namespaces are read-only, as mandated by the Java EE Platform Specification).

```
InitialContext initialContext = new InitialContext();
  initialContext.bind("java:global/a", 100);
```

There is no need to unbind entries created programatically, since WildFly tracks which bindings belong to a deployment, and the bindings are automatically removed when the deployment is undeployed.

WildFly Modules and Extensions

With respect to code in WildFly Modules/Extensions, which is executed out of a Java EE application context, using the standard JNDI API may result in a UnsupportedOperationException if the target namespace uses a WritableServiceBasedNamingStore. To work around that, the bind() invocation needs to be wrapped using WildFly proprietary APIs:

```
InitialContext initialContext = new InitialContext();
  WritableServiceBasedNamingStore.pushOwner(serviceTarget);
  try {
    initialContext.bind("java:global/a", 100);
  } finally {
    WritableServiceBasedNamingStore.popOwner();
  }
```

The ServiceTarget removes the bind when uninstalled, thus using one out of the module/extension domain usage should be avoided, unless entries are removed using unbind().

Naming Subsystem Configuration

It is also possible to bind to one of the three global namespaces using configuration in the naming subsystem. This can be done by either editing the `standalone.xml/domain.xml` file directly, or through the management API.

Four different types of bindings are supported:

- Simple - A primitive or java.net.URL entry (default is `java.lang.String`).
- Object Factory - This allows to to specify the `javax.naming.spi.ObjectFactory` that is used to create the looked up value.
- External Context - An external context to federate, such as an LDAP Directory Service
- Lookup - The allows to create JNDI aliases, when this entry is looked up it will lookup the target and return the result.

An example standalone.xml might look like:

```
<subsystem xmlns="urn:jboss:domain:naming:2.0" >
  <bindings>
    <simple name="java:global/a" value="100" type="int" />
    <simple name="java:global/jbossDocs" value="https://docs.jboss.org" type="java.net.URL" />
    <object-factory name="java:global/b" module="com.acme" class="org.acme.MyObjectFactory" />
    <external-context name="java:global/federation/ldap/example”
class="javax.naming.directory.InitialDirContext" cache="true">
      <environment>
        <property name="java.naming.factory.initial" value=“com.sun.jndi.ldap.LdapCtxFactory” />
        <property name="java.naming.provider.url" value=“ldap://ldap.example.com:389” />
        <property name="java.naming.security.authentication" value=“simple” />
        <property name="java.naming.security.principal" value=“uid=admin,ou=system” />
        <property name="java.naming.security.credentials" value=“secret” />
      </environment>
    </external-context>
    <lookup name="java:global/c" lookup="java:global/b" />
 </bindings>
</subsystem>
```

The CLI may also be used to bind an entry. As an example:

```
/subsystem=naming/binding=java\:global\/mybinding:add(binding-type=simple, type=long,
value=1000)
```

WildFly's Administrator Guide includes a section describing in detail the Naming subsystem configuration.

Retrieving entries from JNDI

Resource Injection

For Java EE applications the recommended way to lookup a JNDI entry is to use `@Resource` injection:

```
@Resource(lookup = "java:global/mystring")
  private String myString;

  @Resource(name = "hello")
  private String hello;

  @Resource
  ManagedExecutorService executor;
```

Note that `@Resource` is more than a JNDI lookup, it also binds an entry in the component's JNDI environment. The new bind JNDI name is defined by `@Resource`'s `name` attribute, which value, if unspecified, is the Java type concatenated with `/` and the field's name, for instance `java.lang.String/myString`. More, similar to when using deployment descriptors to bind JNDI entries. unless the name is an absolute JNDI name, it is considered relative to `java:comp/env`. For instance, with respect to the field named `myString` above, the `@Resource`'s `lookup` attribute instructs WildFly to lookup the value in `java:global/mystring`, bind it in `java:comp/env/java.lang.String/myString`, and then inject such value into the field.

With respect to the field named `hello`, there is no `lookup` attribute value defined, so the responsibility to provide the entry's value is delegated to the deployment descriptor. Considering that the deployment descriptor was the `web.xml` previously shown, which defines an environment entry with same `hello` name, then WildFly inject the valued defined in the deployment descriptor into the field.

The `executor` field has no attributes specified, so the bind's name would default to `java:comp/env/javax.enterprise.concurrent.ManagedExecutorService/executor`, but there is no such entry in the deployment descriptor, and when that happens it's up to WildFly to provide a default value or null, depending on the field's Java type. In this particular case WildFly would inject the default instance of a managed executor service, the value in `java:comp/DefaultManagedExecutorService`, as mandated by the EE Concurrency Utilities 1.0 Specification (JSR 236).

Standard Java SE JNDI API

Java EE applications may use, without any additional configuration needed, the standard JNDI API to lookup an entry from JNDI:

```
String myString = (String) new InitialContext().lookup("java:global/mystring");
```

or simply

```
String myString = InitialContext.doLookup("java:global/mystring");
```

6.28.5 Remote JNDI Reference

Remote JNDI

WildFly supports two different types of remote JNDI. The old jnp based JNDI implementation used in JBoss AS versions prior to 7.x is no longer supported.

remote:

The `remote:` protocol uses the WildFly remoting protocol to lookup items from the servers local JNDI. To use it, you must have the appropriate jars on the class path, if you are maven user can be done simply by adding the following to your `pom.xml`:

```
<dependency>
  <groupId>org.wildfly</groupId>
  <artifactId>wildfly-ejb-client-bom</artifactId>
  <version>8.0.0.Final</version>
  <type>pom</type>
  <scope>compile</scope>
</dependency>
```

If you are not using maven a shaded jar that contains all required classes
can be found in the `bin/client` directory of WildFly's distribution.

```
final Properties env = new Properties();
env.put(Context.INITIAL_CONTEXT_FACTORY,
org.jboss.naming.remote.client.InitialContextFactory.class.getName());
env.put(Context.PROVIDER_URL, "remote://localhost:4447");
remoteContext = new InitialContext(env);
```

ejb:

The ejb: namespace is provided by the jboss-ejb-client library. This protocol allows you to look up EJB's, using their application name, module name, ejb name and interface type.

This is a client side JNDI implementation. Instead of looking up an EJB on the server the lookup name contains enough information for the client side library to generate a proxy with the EJB information. When you invoke a method on this proxy it will use the current EJB client context to perform the invocation. If the current context does not have a connection to a server with the specified EJB deployed then an error will occur. Using this protocol it is possible to look up EJB's that do not actually exist, and no error will be thrown until the proxy is actually used. The exception to this is stateful session beans, which need to connect to a server when they are created in order to create the session bean instance on the server.

Some examples are:

```
ejb:myapp/myejbjar/MyEjbName!com.test.MyRemoteInterface
ejb:myapp/myejbjar/MyStatefulName!comp.test.MyStatefulRemoteInterface?stateful
```

The first example is a lookup of a singleton, stateless or EJB 2.x home interface. This lookup will not hit the server, instead a proxy will be generated for the remote interface specified in the name. The second example is for a stateful session bean, in this case the JNDI lookup will hit the server, in order to tell the server to create the SFSB session.

For more details on how the server connections are configured, please see EJB invocations from a remote client using JNDI.

Remote JNDI Access

WildFly supports two different types of remote JNDI.

http-remoting:

The `http-remoting:` protocol implementation is provided by JBoss Remote Naming project, and uses http upgrade to lookup items from the servers local JNDI. To use it, you must have the appropriate jars on the class path, if you are maven user can be done simply by adding the following to your `pom.xml` dependencies:

```
<dependency>
  <groupId>org.wildfly</groupId>
  <artifactId>wildfly-ejb-client-bom</artifactId>
  <version>8.0.0.Final</version>
  <type>pom</type>
</dependency>
```

If you are not using maven a shaded jar that contains all required classes can be found in the `bin/client` directory of WildFly's distribution.

```
final Properties env = new Properties();
env.put(Context.INITIAL_CONTEXT_FACTORY,
"org.jboss.naming.remote.client.InitialContextFactory");
env.put(Context.PROVIDER_URL, "http-remoting://localhost:8080");
// the property below is required ONLY if there is no ejb client configuration loaded (such as a
// jboss-ejb-client.properties in the class path) and the context will be used to lookup EJBs
env.put("jboss.naming.client.ejb.context", true);
InitialContext remoteContext = new InitialContext(env);
RemoteCalculator ejb = (RemoteCalculator)
remoteContext.lookup("wildfly-http-remoting-ejb/CalculatorBean!"
                + RemoteCalculator.class.getName());
```

The http-remoting client assumes JNDI names in remote lookups are relative to java:jboss/exported namespace, a lookup of an absolute JNDI name will fail.

ejb:

The ejb: namespace implementation is provided by the jboss-ejb-client library, and allows the lookup of EJB's using their application name, module name, ejb name and interface type. To use it, you must have the appropriate jars on the class path, if you are maven user can be done simply by adding the following to your `pom.xml` dependencies:

```
<dependency>
  <groupId>org.wildfly</groupId>
  <artifactId>wildfly-ejb-client-bom</artifactId>
  <version>8.0.0.Final</version>
  <type>pom</type>
</dependency>
```

If you are not using maven a shaded jar that contains all required classes can be found in the `bin/client` directory of WildFly's distribution.

This is a client side JNDI implementation. Instead of looking up an EJB on the server the lookup name contains enough information for the client side library to generate a proxy with the EJB information. When you invoke a method on this proxy it will use the current EJB client context to perform the invocation. If the current context does not have a connection to a server with the specified EJB deployed then an error will occur. Using this protocol it is possible to look up EJB's that do not actually exist, and no error will be thrown until the proxy is actually used. The exception to this is stateful session beans, which need to connect to a server when they are created in order to create the session bean instance on the server.

```
final Properties env = new Properties();
env.put(Context.URL_PKG_PREFIXES, "org.jboss.ejb.client.naming");
InitialContext remoteContext = new InitialContext(env);
MyRemoteInterface myRemote = (MyRemoteInterface)
remoteContext.lookup("ejb:myapp/myejbjar/MyEjbName\!com.test.MyRemoteInterface");
MyStatefulRemoteInterface myStatefulRemote = (MyStatefulRemoteInterface)
remoteContext.lookup("ejb:myapp/myejbjar/MyStatefulName\!comp.test.MyStatefulRemoteInterface?state
```

The first example is a lookup of a singleton, stateless or EJB 2.x home interface. This lookup will not hit the server, instead a proxy will be generated for the remote interface specified in the name. The second example is for a stateful session bean, in this case the JNDI lookup will hit the server, in order to tell the server to create the SFSB session.

For more details on how the server connections are configured, including the **required** jboss ejb client setup, please see EJB invocations from a remote client using JNDI.

6.29 JPA Reference Guide

6.29.1 Introduction

The WildFly JPA subsystem implements the JPA 2.1 container-managed requirements. Deploys the persistence unit definitions, the persistence unit/context annotations and persistence unit/context references in the deployment descriptor. JPA Applications use the Hibernate (version 5.1) persistence provider, which is included with WildFly. The JPA subsystem uses the standard SPI (javax.persistence.spi.PersistenceProvider) to access the Hibernate persistence provider and some additional extensions as well.

During application deployment, JPA use is detected (e.g. persistence.xml or @PersistenceContext/Unit annotations) and injects Hibernate dependencies into the application deployment. This makes it easy to deploy JPA applications.

In the remainder of this documentation, ”entity manager” refers to an instance of the *javax.persistence.EntityManager* class. Javadoc for the JPA interfaces and JPA 2.1 specification.

The index of the Hibernate documentation is at http://hibernate.org/orm/documentation/5.1/.

6.29.2 Update your Persistence.xml for Hibernate 5.1

The persistence provider class name in Hibernate 4.3.0 (and greater) is **org.hibernate.jpa.HibernatePersistenceProvider**.

Instead of specifying:

<provider>org.hibernate.ejb.HibernatePersistence</provider>

Switch to:

<provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>

Or remove the persistence provider class name from your persistence.xml (so the default provider will be used).

6.29.3 Entity manager

The entity manager (javax.persistence.EntityManager class) is similar to the Hibernate Session class; applications use it to create/read/update/delete data (and related operations). Applications can use application-managed or container-managed entity managers. Keep in mind that the entity manager is not thread safe, don't share the same entity manager instance with multiple threads.

Internally, the entity manager, has a persistence context for managing entities. You can think of the persistence context as being closely associated with the entity manager.

6.29.4 Container-managed entity manager

When you inject a container-managed entity managers into an application variable, it is treated like an (EE container controlled) Java proxy object, that will be associated with an underlying EntityManager instance, for each started JTA transaction and is flushed/closed when the JTA transaction commits. Such that when your application code invokes EntityManager.anyMethod(), the current JTA transaction is searched (using persistence unit name as key) for the underlying EntityManager instance, if not found, a new EntityManager instance is created and associated with the current JTA transaction, to be reused for the next EntityManager invocation. Use the @PersistenceContext annotation, to inject a container-managed entity manager into a javax.persistence.EntityManager variable.

6.29.5 Application-managed entity manager

An application-managed entity manager is kept around until the application closes it. The scope of the application-managed entity manager is from when the application creates it and lasts until the application closes it. Use the *@PersistenceUnit* annotation, to inject a persistence unit into a *javax.persistence.EntityManagerFactory variable*. The EntityManagerFactory can return an application-managed entity manager.

6.29.6 Persistence Context

The JPA persistence context contains the entities managed by the entity manager (via the JPA persistence provider). The underlying entity manager maintains the persistence context. The persistence context acts like a first level (transactional) cache for interacting with the datasource. Loaded entities are placed into the persistence context before being returned to the application. Entities changes are also placed into the persistence context (to be saved in the database when the transaction commits).

6.29.7 Transaction-scoped Persistence Context

The transaction-scoped persistence context coordinates with the (active) JTA transaction. When the transaction commits, the persistence context is flushed to the datasource (entity objects are detached but may still be referenced by application code). All entity changes that are expected to be saved to the datasource, must be made during a transaction. Entities read outside of a transaction will be detached when the entity manager invocation completes. Example transaction-scoped persistence context is below.

```
@Stateful  // will use container managed transactions
public class CustomerManager {
  @PersistenceContext(unitName = "customerPU") // default type is
PersistenceContextType.TRANSACTION
  EntityManager em;
  public customer createCustomer(String name, String address) {
    Customer customer = new Customer(name, address);
    em.persist(customer);  // persist new Customer when JTA transaction completes (when method
ends).
                           // internally:
                           //    1. Look for existing "customerPU" persistence context in active
JTA transaction and use if found.
                           //    2. Else create new "customerPU" persistence context (e.g.
instance of org.hibernate.ejb.HibernatePersistence)
                           //       and put in current active JTA transaction.
    return customer;       // return Customer entity (will be detached from the persistence
context when caller gets control)
  }  // Transaction.commit will be called, Customer entity will be persisted to the database and
"customerPU" persistence context closed
```

6.29.8 Extended Persistence Context

The (ee container managed) extended persistence context can span multiple transactions and allows data modifications to be queued up (like a shopping cart), without an active JTA transaction (to be applied during the next JTA TX). The Container-managed extended persistence context can only be injected into a stateful session bean. You can also think of the extended persistence context, as being an entity manager.

```
@PersistenceContext(type = PersistenceContextType.EXTENDED, unitName = "inventoryPU")
EntityManager em;
```

Extended Persistence Context Inheritance

```
JPA 2.0 specification section 7.6.2.1

If a stateful session bean instantiates a stateful session bean (executing in the same EJB
container instance) which also has such an extended persistence context, the extended
persistence context of the first stateful session bean is inherited by the second stateful
session bean and bound to it, and this rule recursively applies—independently of whether
transactions are active or not at the point of the creation of the stateful session beans.
```

By default, the current stateful session bean being created, will (**deeply**) inherit the extended persistence context from any stateful session bean executing in the current Java thread. The **deep** inheritance of extended persistence context includes walking multiple levels up the stateful bean call stack (inheriting from parent beans). The **deep** inheritance of extended persistence context includes sibling beans. For example, parentA references child beans beanBwithXPC & beanCwithXPC. Even though parentA doesn't have an extended persistence context, beanBwithXPC & beanCwithXPC will share the same extended persistence context.

Some other EE application servers, use **shallow** inheritance, where stateful session bean only inherit from the parent stateful session bean (if there is a parent bean). Sibling beans do not share the same extended persistence context unless their (common) parent bean also has the same extended persistence context.

Applications can include a (top-level) **jboss-all.xml** deployment descriptor that specifies either the (default) **DEEP** extended persistence context inheritance or **SHALLOW**.

The WF/docs/schema/jboss-jpa_1_0.xsd describes the **jboss-jpa** deployment descriptor that may be included in the **jboss-all.xml**. Below is an example of using **SHALLOW** extended persistence context inheritance:

```
<jboss>
   <jboss-jpa xmlns="http://www.jboss.com/xml/ns/javaee">
   <extended-persistence inheritance="SHALLOW"/>
   </jboss-jpa>
</jboss>
```

Below is an example of using **DEEP** extended persistence inheritance:

```
<jboss>
   <jboss-jpa xmlns="http://www.jboss.com/xml/ns/javaee">
   <extended-persistence inheritance="DEEP"/>
   </jboss-jpa>
</jboss>
```

The AS console/cli can change the **default** extended persistence context setting (DEEP or SHALLOW). The following cli commands will read the current JPA settings and enable SHALLOW extended persistence context inheritance for applications that do not include the **jboss-jpa** deployment descriptor:

> *./jboss-cli.sh*
> *cd subsystem=jpa*
> *:read-resource*
> *:write-attribute(name=default-extended-persistence-inheritance,value="SHALLOW")*

6.29.9 Entities

JPA allows use of your (pojo) plain old Java class to represent a database table row.

```
@PersistenceContext EntityManager em;
Integer bomPk = getIndexKeyValue();
BillOfMaterials bom = em.find(BillOfMaterials.class, bomPk); // read existing table row into
BillOfMaterials class

BillOfMaterials createdBom = new BillOfMaterials("...");     // create new entity
em.persist(createdBom);  // createdBom is now managed and will be saved to database when the
current JTA transaction completes
```

The entity lifecycle is managed by the underlying persistence provider.

- New (transient): an entity is new if it has just been instantiated using the new operator, and it is not associated with a persistence context. It has no persistent representation in the database and no identifier value has been assigned.
- Managed (persistent): a managed entity instance is an instance with a persistent identity that is currently associated with a persistence context.
- Detached: the entity instance is an instance with a persistent identity that is no longer associated with a persistence context, usually because the persistence context was closed or the instance was evicted from the context.
- Removed: a removed entity instance is an instance with a persistent identity, associated with a persistence context, but scheduled for removal from the database.

6.29.10 Deployment

The persistence.xml contains the persistence unit configuration (e.g. datasource name) and as described in the JPA 2.0 spec (section 8.2), the jar file or directory whose META-INF directory contains the persistence.xml file is termed the root of the persistence unit. In Java EE environments, the root of a persistence unit must be one of the following (quoted directly from the JPA 2.0 specification):

"

- an EJB-JAR file
- the WEB-INF/classes directory of a WAR file
- a jar file in the WEB-INF/lib directory of a WAR file
- a jar file in the EAR library directory
- an application client jar file

The persistence.xml can specify either a JTA datasource or a non-JTA datasource. The JTA datasource is expected to be used within the EE environment (even when reading data without an active transaction). If a datasource is not specified, the default-datasource will instead be used (must be configured).

NOTE: Java Persistence 1.0 supported use of a jar file in the root of the EAR as the root of a persistence unit. This use is no longer supported. Portable applications should use the EAR library directory for this case instead.

"

Question: Can you have a EAR/META-INF/persistence.xml?

Answer: No, the above may deploy but it could include other archives also in the EAR, so you may have deployment issues for other reasons. Better to put the persistence.xml in an EAR/lib/somePuJar.jar.

6.29.11 Troubleshooting

The **org.jboss.as.jpa** logging can be enabled to get the following information:

- INFO - when persistence.xml has been parsed, starting of persistence unit service (per deployed persistence.xml), stopping of persistence unit service
- DEBUG - informs about entity managers being injected, creating/reusing transaction scoped entity manager for active transaction
- TRACE - shows how long each entity manager operation took in milliseconds, application searches for a persistence unit, parsing of persistence.xml

To enable TRACE, open the as/standalone/configuration/standalone.xml (or as/domain/configuration/domain.xml) file. Search for **<subsystem xmlns="urn:jboss:domain:logging:1.0">** and add the **org.jboss.as.jpa** category. You need to change the console-handler level from **INFO** to **TRACE**.

```
<subsystem xmlns="urn:jboss:domain:logging:1.0">
     <console-handler name="CONSOLE">
      <level name="TRACE" />
      ...
     </console-handler>

     </periodic-rotating-file-handler>
     <logger category="com.arjuna">
        <level name="WARN" />
     </logger>

     <logger category="org.jboss.as.jpa">
        <level name="TRACE" />
     </logger>

     <logger category="org.apache.tomcat.util.modeler">
        <level name="WARN" />
     </logger>
     ...
```

To see what is going on at the JDBC level, enable **jboss.jdbc.spy** TRACE and add spy="true" to the datasource.

```
<datasource jndi-name="java:jboss/datasources/..." pool-name="..." enabled="true" spy="true">
<logger category="jboss.jdbc.spy">
  <level name="TRACE"/>
</logger>
```

To troubleshoot issues with the Hibernate second level cache, try enabling trace for **org.hibernate.SQL + org.hibernate.cache.infinispan + org.infinispan:**

```
<subsystem xmlns="urn:jboss:domain:logging:1.0">
     <console-handler name="CONSOLE">
      <level name="TRACE" />
      ...
     </console-handler>

     </periodic-rotating-file-handler>
     <logger category="com.arjuna">
        <level name="WARN" />
     </logger>

     <logger category="org.hibernate.SQL">
        <level name="TRACE" />
     </logger>

     <logger category="org.hibernate">
        <level name="TRACE" />
     </logger>
      <logger category="org.infinispan">
        <level name="TRACE" />
     </logger>

     <logger category="org.apache.tomcat.util.modeler">
        <level name="WARN" />
     </logger>
     ...
```

6.29.12 Using the Infinispan second level cache

To enable the second level cache with Hibernate 5.1, just set the **hibernate.cache.use_second_level_cache** property to true, as is done in the following example (also set the shared-cache-mode accordingly). By default the application server uses Infinispan as the cache provider for **JPA applications**, so you don't need specify anything on top of that. The Infinispan version that is included in WildFly is expected to work with the Hibernate version that is included with WildFly. Example persistence.xml settings:

```
<?xml version="1.0" encoding="UTF-8"?><persistence
xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">
<persistence-unit name="2lc_example_pu">
   <description>example of enabling the second level cache.</description>
   <jta-data-source>java:jboss/datasources/mydatasource</jta-data-source>
   <shared-cache-mode>ENABLE_SELECTIVE</shared-cache-mode>
   <properties>
      <property name="hibernate.cache.use_second_level_cache" value="true"/>
   </properties>
</persistence-unit>
</persistence>
```

Here is an example of enabling the second level cache for a Hibernate native API hibernate.cfg.xml file:

```
<property name="hibernate.cache.region.factory_class"
value="org.jboss.as.jpa.hibernate5.infinispan.InfinispanRegionFactory"/>
<property name="hibernate.cache.infinispan.cachemanager"
value="java:jboss/infinispan/container/hibernate"/>
<property name="hibernate.transaction.manager_lookup_class"
value="org.hibernate.transaction.JBossTransactionManagerLookup"/>
<property name="hibernate.cache.use_second_level_cache" value="true"/>
```

The Hibernate native API application will also need a MANIFEST.MF:

```
Dependencies: org.infinispan,org.hibernate
```

Infinispan Hibernate/JPA second level cache provider documentation contains advanced configuration information but you should bear in mind that when Hibernate runs within WildFly 8, some of those configuration options, such as region factory, are not needed. Moreover, the application server providers you with option of selecting a different cache container for Infinispan via **hibernate.cache.infinispan.container** persistence property. To reiterate, this property is not mandatory and a default container is already deployed for by the application server to host the second level cache.

Here is an example of what the Hibernate cache settings may currently be in your standalone.xml:

```
<cache-container name="hibernate" default-cache="local-query" module="org.hibernate.infinispan">
    <local-cache name="entity">
        <transaction mode="NON_XA"/>
        <eviction strategy="LRU" max-entries="10000"/>
        <expiration max-idle="100000"/>
    </local-cache>
    <local-cache name="local-query">
        <eviction strategy="LRU" max-entries="10000"/>
        <expiration max-idle="100000"/>
    </local-cache>
    <local-cache name="timestamps"/>
</cache-container>
```

Below is an example of customizing the "entity", "immutable-entity", "local-query", "pending-puts", "timestamps" cache configuration may look like:

```
<cache-container name="hibernate" module="org.hibernate.infinispan"
default-cache="immutable-entity">
    <local-cache name="entity">
        <transaction mode="NONE"/>
        <eviction max-entries="-1"/>
        <expiration max-idle="120000"/>
    </local-cache>
    <local-cache name="immutable-entity">
        <transaction mode="NONE"/>
        <eviction max-entries="-1"/>
        <expiration max-idle="120000"/>
    </local-cache>
    <local-cache name="local-query">
        <eviction max-entries="-1"/>
        <expiration max-idle="300000"/>
    </local-cache>
    <local-cache name="pending-puts">
        <transaction mode="NONE"/>
        <eviction strategy="NONE"/>
        <expiration max-idle="60000"/>
    </local-cache>
    <local-cache name="timestamps">
        <transaction mode="NONE"/>
        <eviction strategy="NONE"/>
    </local-cache>
</cache-container>
```

Persistence.xml to use the above custom settings:

```
<properties>
    <property name="hibernate.cache.use_second_level_cache" value="true"/>
    <property name="hibernate.cache.use_query_cache" value="true"/>
    <property name="hibernate.cache.infinispan.immutable-entity.cfg" value="immutable-entity"/>
    <property name="hibernate.cache.infinispan.timestamps.cfg" value="timestamps"/>
    <property name="hibernate.cache.infinispan.pending-puts.cfg" value="pending-puts"/>
</properties>
```

6.29.13 Replacing the current Hibernate 5.x jars with a newer version

Just update the current wildfly/modules/system/layers/base/org/hibernate/main folder to contain the newer version (after stopping your WildFly server instance).

1. Delete *.index files in wildfly/modules/system/layers/base/org/hibernate/main and wildfly/modules/system/layers/base/org/hibernate/envers/main folders.
2. Backup the current contents of wildfly/modules/system/layers/base/org/hibernate in case you make a mistake.
3. Remove the older jars and copy new Hibernate jars into wildfly/modules/system/layers/base/org/hibernate/main + wildfly/modules/system/layers/base/org/hibernate/envers/main.
4. Update the wildfly/modules/system/layers/base/org/hibernate/main/module.xml + wildfly/modules/system/layers/base/org/hibernate/envers/main/module.xml to name the jars that you copied in.
5. Also update the hibernate-infinispan jars in wildfly/modules/system/layers/base/org/hibernate/infinispan.

6.29.14 Using Hibernate Search

WildFly includes Hibernate Search. If you want to use the bundled version of Hibernate Search - which requires to use the default Hibernate ORM 5.1 persistence provider - this will be automatically enabled. Having this enabled means that, provided your application includes any entity which is annotated with **org.hibernate.search.annotations.Indexed**, the module **org.hibernate.search.orm:main** will be made available to your deployment; this will also include the required version of Apache Lucene.

If you do not want this module to be exposed to your deployment, set the persistence property **wildfly.jpa.hibernate.search.module** to either **none** to not automatically inject any Hibernate Search module, or to any other module identifier to inject a different module.
For example you could set **wildfly.jpa.hibernate.search.module=org.hibernate.search.orm:5.4.0.Alpha1** to use the experimental version 5.4.0.Alpha1 instead of the provided module; in this case you'll have to download and add the custom modules to the application server as other versions are not included.
When setting **wildfly.jpa.hibernate.search.module=none** you might also opt to include Hibernate Search and its dependencies within your application but we highly recommend the modules approach.

6.29.15 Packaging the Hibernate JPA persistence provider with your application

WildFly allows the packaging of Hibernate persistence provider jars with the application. The JPA deployer will detect the presence of a persistence provider in the application and **jboss.as.jpa.providerModule** needs to be set to **application**.<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">
<persistence-unit name="myOwnORMVersion_pu">
<description>Hibernate Persistence Unit.</description>
<jta-data-source>java:jboss/datasources/PlannerDS</jta-data-source>
<properties>
   <property name="jboss.as.jpa.providerModule" value="application" />
</properties>
</persistence-unit>
</persistence>

6.29.16 Migrating from OpenJPA

You need to copy the OpenJPA jars (e.g. openjpa-all.jar serp.jar) into the WildFly modules/org/apache/openjpa/main folder and update modules/org/apache/openjpa/main/module.xml to include the same jar file names that you copied in. This will help you get your application that depends on OpenJPA, to deploy on WildFly.

```
<module xmlns="urn:jboss:module:1.1" name="org.apache.openjpa">
    <resources>
        <resource-root path="jipijapa-openjpa-1.0.1.Final.jar"/>
        <resource-root path="openjpa-all.jar">
           <filter>
              <exclude path="javax/**" />
           </filter>
        </resource-root>
        <resource-root path="serp.jar"/>
    </resources>

    <dependencies>
        <module name="javax.api"/>
        <module name="javax.annotation.api"/>
        <module name="javax.enterprise.api"/>
        <module name="javax.persistence.api"/>
        <module name="javax.transaction.api"/>
        <module name="javax.validation.api"/>
        <module name="javax.xml.bind.api"/>
        <module name="org.apache.commons.collections"/>
        <module name="org.apache.commons.lang"/>
        <module name="org.jboss.as.jpa.spi"/>
        <module name="org.jboss.logging"/>
        <module name="org.jboss.vfs"/>
        <module name="org.jboss.jandex"/>
    </dependencies>
</module>
```

6.29.17 Migrating from EclipseLink

You need to copy the EclipseLink jar (e.g. eclipselink-2.6.0.jar or eclipselink.jar as in the example below) into the WildFly modules/org/eclipse/persistence/main folder and update modules/org/eclipse/persistence/main/module.xml to include the EclipseLink jar (take care to use the jar name that you copied in). If you happen to leave the EclipseLink version number in the jar name, the module.xml should reflect that. This will help you get your application that depends on EclipseLink, to deploy on WildFly.

```
<module xmlns="urn:jboss:module:1.1" name="org.eclipse.persistence">
    <resources>
        <resource-root path="jipijapa-eclipselink-10.0.0.Final.jar"/>
        <resource-root path="eclipselink.jar">           <filter>
              <exclude path="javax/**" />
           </filter>
        </resource-root>
    </resources>

    <dependencies>
        <module name="asm.asm"/>
        <module name="javax.api"/>
        <module name="javax.annotation.api"/>
        <module name="javax.enterprise.api"/>
        <module name="javax.persistence.api"/>
        <module name="javax.transaction.api"/>
        <module name="javax.validation.api"/>
        <module name="javax.xml.bind.api"/>
        <module name="org.antlr"/>
        <module name="org.apache.commons.collections"/>
        <module name="org.dom4j"/>
        <module name="org.jboss.as.jpa.spi"/>
        <module name="org.jboss.logging"/>
        <module name="org.jboss.vfs"/>
    </dependencies>
</module>
```

As a workaround for issueid=414974, set (WildFly) system property "eclipselink.archive.factory" to value "org.jipijapa.eclipselink.JBossArchiveFactoryImpl" via jboss-cli.sh command (WildFly server needs to be running when this command is issued):

```
jboss-cli.sh --connect
'/system-property=eclipselink.archive.factory:add(value=org.jipijapa.eclipselink.JBossArchiveFacto
```

. The following shows what the standalone.xml (or your WildFly configuration you are using) file might look like after updating the system properties:

```
<system-properties>
   ...
   <property name="eclipselink.archive.factory"
value="org.jipijapa.eclipselink.JBossArchiveFactoryImpl"/>
</system-properties>
```

You should then be able to deploy applications with persistence.xml that include;

```
<provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
```

Also refer to page how to use EclipseLink with WildFly guide here.

6.29.18 Migrating from DataNucleus

Read the how to use DataNucleus with WildFly guide here.

6.29.19 Native Hibernate use

Applications that use the Hibernate API directly, are referred to here as native Hibernate applications. Native Hibernate applications, can choose to use the Hibernate jars included with WildFly or they can package their own copy of the Hibernate jars. Applications that utilize JPA will automatically have the Hibernate classes injected onto the application deployment classpath. Meaning that JPA applications, should expect to use the Hibernate jars included in WildFly.

Example MANIFEST.MF entry to add dependency for Hibernate native applications:

```
Manifest-Version: 1.0
...
Dependencies: org.hibernate
```

If you use the Hibernate native api in your application and also use the JPA api to access the same entities (from the same Hibernate session/EntityManager), you could get surprising results (e.g. HibernateSession.saveOrUpdate(entity) is different than EntityManager.merge(entity). Each entity should be managed by either Hibernate native API or JPA code.

6.29.20 Injection of Hibernate Session and SessionFactoryInjection of Hibernate Session and SessionFactory

You can inject a org.hibernate.Session and org.hibernate.SessionFactory directly, just as you can do with EntityManagers and EntityManagerFactorys.

```
import org.hibernate.Session;
import org.hibernate.SessionFactory;
@Stateful public class MyStatefulBean ... {
   @PersistenceContext(unitName="crm") Session session1;
   @PersistenceContext(unitName="crm2", type=EXTENDED) Session extendedpc;
   @PersistenceUnit(unitName="crm") SessionFactory factory;
}
```

6.29.21 Hibernate properties

WildFly automatically sets the following Hibernate (5.x) properties (if not already set in persistence unit definition):

Property	Purpose
hibernate.id.new_generator_mappings =true	New applications should let th default to true, older applicatio with existing data might need set to false (see note below). really depends on whether yo application uses the @GeneratedValue(AUTO) wh will generates new key values newly created entities. The application can override this value (in the persistence.xml)
hibernate.transaction.jta.platform= instance of org.hibernate.service.jta.platform.spi.JtaPlatform interface	The transaction manager, use transaction and transaction synchronization registry is passed into Hibernate via this class.
hibernate.ejb.resource_scanner = instance of org.hibernate.ejb.packaging.Scanner interface	Instance of entity scanning cla is passed in that knows how t use the AS annotation indexe (for faster deployment).
hibernate.transaction.manager_lookup_class	This property is removed if fou in the persistence.xml (could conflict with JtaPlatform)
hibernate.session_factory_name = qualified persistence unit name	Is set to the application name persistence unit name (application can specify a different value but it needs to unique across all application deployments on the AS instance).
hibernate.session_factory_name_is_jndi = false	only set if the application didn specify a value for hibernate.session_factory_na

hibernate.ejb.entitymanager_factory_name = qualified persistence unit name	Is set to the application name persistence unit name (application can specify a different value but it needs to unique across all application deployments on the AS instance).
hibernate.query.jpaql_strict_compliance=true	
hibernate.auto_quote_keyword=false	
hibernate.implicit_naming_strategy =org.hibernate.boot.model.naming.ImplicitNamingStrategyJpaCompliantImpl	

In Hibernate 4.x (and greater), if **new_generator_mappings** is **true**:

- @GeneratedValue(AUTO) maps to org.hibernate.id.enhanced.SequenceStyleGenerator
- @GeneratedValue(TABLE) maps to org.hibernate.id.enhanced.TableGenerator
- @GeneratedValue(SEQUENCE) maps to org.hibernate.id.enhanced.SequenceStyleGenerator

In Hibernate 4.x (and greater), if **new_generator_mappings** is **false**:

- @GeneratedValue(AUTO) maps to Hibernate "native"
- @GeneratedValue(TABLE) maps to org.hibernate.id.MultipleHiLoPerTableGenerator
- @GeneratedValue(SEQUENCE) to Hibernate "seqhilo"

6.29.22 Persistence unit properties

The following properties are supported in the persistence unit definition (in the persistence.xml file):

Property	Purpose
jboss.as.jpa.providerModule	name of the persistence provider module (default is **org.hibernate**). Should be **application**, if a persistence provider is packaged with the application. See note below about some module names that are built in (based on the **provider**).
jboss.as.jpa.adapterModule	name of the integration classes that help WildFly to work with the persistence provider.
jboss.as.jpa.adapterClass	class name of the integration adapter.
jboss.as.jpa.managed	set to **false** to disable container managed JPA access to the persistence unit. The default is **true**, which enables container managed JPA access to the persistence unit. This is typically set to **false** for Spring applications.

jboss.as.jpa.classtransformer	set to **false** to disable class transformers for the persistence unit. Set to **true**, to allow entity class enhancing/rewriting.
wildfly.jpa.default-unit	set to **true** to choose the default persistence unit in an application. This is useful if you inject a persistence context without specifying the unitName (@PersistenceContext EntityManager em) but have multiple persistence units specified in your persistence.xml.
wildfly.jpa.twophasebootstrap	persistence providers (like Hibernate ORM 4.3+ via EntityManagerFactoryBuilder), allow a two phase persistence unit bootstrap, which improves JPA integration with CDI. Setting the **wildfly.jpa.twophasebootstrap** hint to false, disables the two phase bootstrap (for the persistence unit that contains the hint).
wildfly.jpa.allowdefaultdatasourceuse	set to false to prevent persistence unit from using the default data source. Defaults to true. This is only important for persistence units that do not specify a datasource.
jboss.as.jpa.deferdetach	Controls whether transaction scoped persistence context used in non-JTA transaction thread, will detach loaded entities after each EntityManager invocation or when the persistence context is closed (e.g. business method ends). Defaults to false (entities are cleared after EntityManager invocation) and if set to true, the detach is deferred until the context is closed.
wildfly.jpa.hibernate.search.module	Controls which version of Hibernate Search to include on classpath. Only makes sense when using Hibernate as JPA implementation. The default is **auto**; other valid values are **none** or a full module identifier to use an alternative version.
jboss.as.jpa.scopedname	Specify the qualified (application scoped) persistence unit name to be used. By default, this is internally set to the application name + persistence unit name. The hibernate.cache.region_prefix will default to whatever you set jboss.as.jpa.scopedname to. Make sure you set the jboss.as.jpa.scopedname value to a value not already in use by other applications deployed on the same application server instance.

wildfly.jpa.allowjoinedunsync	If set to true, allows an SynchronizationType.UNSYNCHRONIZED persistence context that has been joined to the active JTA transaction, to be propagated into a SynchronizationType.SYNCHRONIZED persistence context. Otherwise, an IllegalStateException exception would of been thrown that complains that an unsychronized persistence context cannot be propagated into a synchronized persistence context. Defaults to false.
wildfly.jpa.skipmixedsynctypechecking	Set to true to disable the throwing of an IllegalStateException exception when propagating an SynchronizationType.UNSYNCHRONIZED persistence context into a SynchronizationType.SYNCHRONIZED persistence context. This is a workaround intended to allow applications that used to incorrectly not get IllegalStateException exception with extended persistence contexts, to avoid the IllegalStateException, so they don't have to change their application right away (for compatibility purposes). This hint may be deprecated in a future release. See WFLY-7108 for more details. Defaults to false.

6.29.23 Determine the persistence provider module

As mentioned above, if the **jboss.as.jpa.providerModule** property is not specified, the provider module name is determined by the **provider** name specified in the persistence.xml. The mapping is:

Provider Name	Module name
blank	**org.hibernate**
org.hibernate.ejb.HibernatePersistence	**org.hibernate**
org.hibernate.ogm.jpa.HibernateOgmPersistence	**org.hibernate.ogm**
oracle.toplink.essentials.PersistenceProvider	**oracle.toplink**
oracle.toplink.essentials.ejb.cmp3.EntityManagerFactoryProvider	**oracle.toplink**
org.eclipse.persistence.jpa.PersistenceProvider	**org.eclipse.persistence**
org.datanucleus.api.jpa.PersistenceProviderImpl	**org.datanucleus**
org.datanucleus.store.appengine.jpa.DatastorePersistenceProvider	**org.datanucleus:appengine**
org.apache.openjpa.persistence.PersistenceProviderImpl	**org.apache.openjpa**

6.29.24 Binding EntityManagerFactory/EntityManager to JNDI

By default WildFly does **not** bind the entity manager factory to JNDI. However, you can explicitly configure this in the persistence.xml of your application by setting the `jboss.entity.manager.factory.jndi.name` hint. The value of that property should be the JNDI name to which the entity manager factory should be bound.

You can also bind a container managed (transaction scoped) entity manager to JNDI as well, }}via hint jboss.entity.manager.jndi.name{}{{. As a reminder, a transaction scoped entity manager (persistence context), acts as a proxy that always gets an unique underlying entity manager (at the persistence provider level).

Here's an example:

persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
   xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
        http://java.sun.com/xml/ns/persistence
        http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
   <persistence-unit name="myPU">
      <!-- If you are running in a production environment, add a managed
         data source, the example data source is just for proofs of concept! -->
      <jta-data-source>java:jboss/datasources/ExampleDS</jta-data-source>
      <properties>
         <!-- Bind entity manager factory to JNDI at java:jboss/myEntityManagerFactory -->
         <property name="jboss.entity.manager.factory.jndi.name"
value="java:jboss/myEntityManagerFactory" />
         <property name="jboss.entity.manager.jndi.name" value="java:/myEntityManager"/>
       </properties>
   </persistence-unit>
</persistence>
```

```
@Stateful
public class ExampleSFSB {
  public void createSomeEntityWithTransactionScopedEM(String name) {
    Context context = new InitialContext();
    javax.persistence.EntityManager entityManager = (javax.persistence.EntityManager)
context.lookup("java:/myEntityManager");
    SomeEntity someEntity = new SomeEntity();
    someEntity.setName(name);    entityManager.persist(name);
  }
}
```

6.29.25 Community

Many thanks to the community, for reporting issues, solutions and code changes. A number of people have been answering Wildfly forum questions related to JPA usage. I would like to thank them for this, as well as those reporting issues. For those of you that haven't downloaded the AS source code and started hacking patches together. I would like to encourage you to start by reading Hacking on WildFly. You will find that it easy very easy to find your way around the WildFly/JPA/* source tree and make changes. Also, new for WildFly, is the JipiJapa project that contains additional integration code that makes EE JPA application deployments work better. The following list of contributors should grow over time, I hope to see more of you listed here.

People who have contributed to the WildFly JPA layer:

- Carlo de Wolf (lead of the EJB3 project)
- Steve Ebersole (lead of the Hibernate ORM project)
- Stuart Douglas (lead of the Seam Persistence project, WildFly project team member/committer)
- Jaikiran Pai (Active member of JBoss forums and JBoss EJB3 project team member)
- Strong Liu (leads the productization effort of Hibernate in the EAP product)
- Scott Marlow (lead of the WildFly container JPA sub-project)
- Antti Laisi **(OpenJPA integration changes)**
- Galder Zamarreño (Infinispan 2lc documentation)
- Sanne Grinovero (lead of the Hibernate Search project)
- Paul Ferraro (Infinispan 2lc integration)

6.30 OSGi

WildFly does not include support for OSGi, such functionality is now responsibility of JBoss OSGi project.

JBoss OSGi 2.5.0.Final will provide OSGi support for WildFly 10.

Release progress can be tracked via JBOSGI-786.

6.31 Remote EJB invocations via JNDI - EJB client API or remote-naming project

6.31.1 Purpose

WildFly provides EJB client API project as well as remote-naming project for invoking on remote objects exposed via JNDI. This article explains which approach to use when and what the differences and scope of each of these projects is.

6.31.2 History

Previous versions of JBoss AS (versions < WildFly 8) used JNP project (http://anonsvn.jboss.org/repos/jbossas/projects/naming/) as the JNDI naming implementation. Developers of client applications of previous versions of JBoss AS will be familiar with the `jnp:// PROVIDER_URL` URL they used to use in their applications for communicating with the JNDI server on the JBoss server.

Starting WildFly 8, the JNP project is *not* used. Neither on the server side nor on the client side. The client side of the JNP project has now been replaced by jboss-remote-naming project (https://github.com/jbossas/jboss-remote-naming). There were various reasons why the JNP client was replaced by jboss-remote-naming project. One of them was the JNP project did not allow fine grained security configurations while communicating with the JNDI server. The jboss-remote-naming project is backed by the jboss-remoting project (https://github.com/jboss-remoting/jboss-remoting) which allows much more and better control over security.

6.31.3 Overview

Now that we know that for remote client JNDI communication with WildFly 8 requires jboss-remote-naming project, let's quickly see what the code looks like.

Client code relying on jndi.properties in classpath

```
void doLookup() {
  // Create an InitialContext using the javax.naming.* API
  Context ctx = new InitialContext();
  ctx.lookup("foo/bar");
  ...
}
```

As you can see, there's not much here in terms of code. We first create a InitialContext (http://download.oracle.com/javase/6/docs/api/javax/naming/InitialContext.html) which as per the API will look for a jndi.properties in the classpath of the application. We'll see what our jndi.properties looks like, later. Once the InitialContext is created, we just use it to do a lookup on a JNDI name which we know is bound on the server side. We'll come back to the details of this lookup string in a while.

Let's now see what the jndi.properties in our client classpath looks like:

```
java.naming.factory.initial=org.jboss.naming.remote.client.InitialContextFactory
java.naming.provider.url=http-remoting://localhost:8080
```

Those 2 properties are important for jboss-remote-naming project to be used for communicating with the WildFly server. The first property tells the JNDI API which initial context factory to use. In this case we are pointing it to the InitailContextFactory class supplied by the jboss-remote-naming project. The other property is the PROVIDER_URL. Developers familiar with previous JBoss AS versions would remember that they used `jnp://localhost:1099` (just an example). In WildFly, the URI protocol scheme for jboss-remote-naming project is `remote://`. The rest of the PROVIDER_URL part is the server hostname or IP and the port on which the remoting connector is exposed on the server side. By default the http-remoting connector port in WildFly 8 is 8080. That's what we have used in our example. The hostname we have used is localhost but that should point to the server IP or hostname where the server is running.

JNP client project in previous AS versions allowed a comma separated list for PROVIDER_URL value, so that if one of the server isn't accessible then the JNDI API would use the next available server. The jboss-remote-naming project has similar support starting 1.0.3.Final version of that project (which is available in a WildFly release **after** 7.1.1.Final).

WildFly 8 can use the PROVIDER_URL like:

java.naming.provider.url=http-remoting://server1:8080,http-remoting://server2:8080

So we saw how to setup the JNDI properties in the jndi.properties file. The JNDI API also allows you to pass these properties to the constructor of the InitialContext class (please check the javadoc of that class for more details). Let's quickly see what the code would look like:

```
void doLookup() {
  Properties jndiProps = new Properties();
  jndiProps.put(Context.INITIAL_CONTEXT_FACTORY,
"org.jboss.naming.remote.client.InitialContextFactory");
  jndiProps.put(Context.PROVIDER_URL,"http-remoting://localhost:8080");
  // create a context passing these properties
  Context ctx = new InitialContext(jndiProps);
  // lookup
  ctx.lookup("foo/bar");
  ...
}
```

That's it! You can see that the values that we pass to those properties are the same as what we did via the jndi.properties. It's upto the client application to decide which approach they want to follow.

How does remoting naming work

We have so far had an overview of how the client code looks like when using the jboss-remote-naming (henceforth referred to as remote-naming - too tired of typing jboss-remote-naming everytime ) project. Let's now have a brief look at how the remote-naming project internally establishes the communication with the server and allows JNDI operations from the client side.

Like previously mentioned, remote-naming internally uses jboss-remoting project. When the client code creates an InitialContext backed by the org.jboss.naming.remote.client.InitialContextFactory class, the `org.jboss.naming.remote.client.InitialContextFactory` internally looks for the PROVIDER_URL (and other) properties that are applicable for that context (*doesn't* matter whether it comes from the jndi.properties file or whether passed explicitly to the constructor of the InitialContext). Once it identifies the server and port to connect to, the remote-naming project internally sets up a connection using the jboss-remoting APIs with the remoting connector which is exposed on that port.

We previously mentioned that remote-naming, backed by jboss-remoting project, has increased support for security configurations. Starting WildFly 8, every service including the http remoting connector (which listens by default on port 8080), is secured (see https://community.jboss.org/wiki/AS710Beta1-SecurityEnabledByDefault for details). This means that when trying to do JNDI operations like a lookup, the client has to pass appropriate user credentials. In our examples so far we haven't passed any username/pass or any other credentials while creating the InitialContext. That was just to keep the examples simple. But let's now take the code a step further and see one of the ways how we pass the user credentials. Let's at the moment just assume that the remoting connector on port 8080 is accessible to a user named "`peter`" whose password is expected to be "`lois`".

Note: The server side configurations for the remoting connector to allow "peter" to access the connector, is out of the scope of this documentation. The WildFly 8 documentation already has chapters on how to set that up.

```
void doLookup() {
  Properties jndiProps = new Properties();
  jndiProps.put(Context.INITIAL_CONTEXT_FACTORY,
"org.jboss.naming.remote.client.InitialContextFactory");
  jndiProps.put(Context.PROVIDER_URL,"http-remoting://localhost:8080");
  // username
  jndiProps.put(Context.SECURITY_PRINCIPAL, "peter");
  // password
  jndiProps.put(Context.SECURITY_CREDENTIALS, "lois");
  // create a context passing these properties
  Context ctx = new InitialContext(jndiProps);
  // lookup
  ctx.lookup("foo/bar");
  ...
}
```

The code is similar to our previous example, except that we now have added 2 additional properties that are passed to the InitialContext constructor. The first is http://docs.oracle.com/javase/6/docs/api/javax/naming/Context.html#SECURITY_PRINCIPAL which passes the username (peter in this case) and the second is http://docs.oracle.com/javase/6/docs/api/javax/naming/Context.html#SECURITY_CREDENTIALS which passes the password (lois in this case). Of course the same properties can be configured in the jndi.properties file (read the javadoc of the Context class for appropriate properties to be used in the jndi.properties). This is one way of passing the security credentials for JNDI communication with WildFly. There are some other ways to do this too. But we won't go into those details here for two reasons. One, it's outside the scope of this article and two (which is kind of the real reason) I haven't looked fully at the remote-naming implementation details to see what other ways are allowed.

JNDI operations allowed using remote-naming project

So far we have mainly concentrated on how the naming context is created and what it internally does when an instance is created. Let's now take this one step further and see what kind of operations are allowed for a JNDI context backed by the remote-naming project.

The JNDI Context has various methods http://docs.oracle.com/javase/6/docs/api/javax/naming/Context.html that are exposed for JNDI operations. One important thing to note in case of remote-naming project is that, the project's scope is to allow a client to communicate with the JNDI backend exposed by the server. As such, the remote-naming project does **not** support many of the methods that are exposed by the javax.naming.Context class. The remote-naming project only supports the read-only kind of methods (like the lookup() method) and does not support any write kind of methods (like the bind() method). The client applications are expected to use the remote-naming project mainly for lookups of JNDI objects. Neither WildFly 8 nor remote-naming project allows writing/binding to the JNDI server from a remote application.

requisites of remotely accessible JNDI objects

On the server side, the JNDI can contain numerous objects that are bound to it. However, *not* all of those are exposed remotely. The two conditions that are to be satisfied by the objects bound to JNDI, to be remotely accessible are:

1) Such objects should be bound under the `java:jboss/exported/` namespace. For example, `java:jboss/exported/foo/bar`
2) Objects bound to the `java:jboss/exported/` namespace are expected to be serializable. This allows the objects to be sent over the wire to the remote clients

Both these conditions are important and are required for the objects to be remotely accessible via JNDI.

JNDI lookup strings for remote clients backed by the remote-naming project

In our examples, so far, we have been consistently using "`foo/bar`" as the JNDI name to lookup from a remote client using the remote-naming project. There's a bit more to understand about the JNDI name and how it maps to the JNDI name that's bound on the server side.

First of all, the JNDI names used while using the remote-naming project are **always** relative to the java:jboss/exported/ namespace. So in our examples, we are using "`foo/bar`" JNDI name for the lookup, that actually is (internally) "`java:jboss/exported/foo/bar`". The remote-naming project expects it to **always** be relative to the "`java:jboss/exported/`" namespace. Once connected with the server side, the remote-naming project will lookup for "foo/bar" JNDI name under the "`java:jboss/exported/`" namespace of the server.

Note: Since the JNDI name that you use on the client side is **always** relative to java:jboss/exported namespace, you **shouldn't** be prefixing the java:jboss/exported/ string to the JNDI name. For example, if you use the following JNDI name:

ctx.lookup("java:jboss/exported/helloworld");

then remote-naming will translate it to

ctx.lookup("java:jboss/exported/java:jboss/exported/helloworld");

and as a result, will fail during lookup.

The remote-naming implementation perhaps should be smart enough to strip off the java:jboss/exported/ namespace prefix if supplied. But let's not go into that here.

How does remote-naming project implementation transfer the JNDI objects to the clients

When a lookup is done on a JNDI string, the remote-naming implementation internally uses the connection to the remoting connector (which it has established based on the properties that were passed to the InitialContext) to communicate with the server. On the server side, the implementation then looks for the JNDI name under the `java:jboss/exported/` namespace. Assuming that the JNDI name is available, under that namespace, the remote-naming implementation then passes over the object bound at that address to the client. This is where the requirement about the JNDI object being serializable comes into picture. remote-naming project internally uses jboss-marshalling project to marshal the JNDI object over to the client. On the client side the remote-naming implementation then unmarshalles the object and returns it to the client application.

So literally, each lookup backed by the remote-naming project entails a server side communication/interaction and then marshalling/unmarshalling of the object graph. This is very important to remember. We'll come back to this later, to see why this is important when it comes to using EJB client API project for doing EJB lookups (EJB invocations from a remote client using JNDI) as against using remote-naming project for doing the same thing.

6.31.4 Summary

That pretty much covers whatever is important to know, in the remote-naming project, for a typical client application. Don't close the browser yet though, since we haven't yet come to the part of EJB invocations from a remote client using the remote-naming project. In fact, the motivation behind writing this article was to explain why *not* to use remote-naming project (in most cases) for doing EJB invocations against WildFly server.

Those of you who don't have client applications doing remote EJB invocations, can just skip the rest of this article if you aren't interested in those details.

6.31.5 Remote EJB invocations backed by the remote-naming project

In previous sections of this article we saw that whatever is exposed in the java:jboss/exported/ namespace is accessible remotely to the client applications under the relative JNDI name. Some of you might already have started thinking about exposing remote views of EJBs under that namespace.

It's important to note that WildFly server side already by default exposes the remote views of a EJB under the `java:jboss/exported/` namespace (although it isn't logged in the server logs). So assuming your server side application has the following stateless bean:

```
package org.myapp.ejb;

@Stateless
@Remote(Foo.class)
public class FooBean implements Foo {
...
 public String sayBar() {
     return "Baaaaaaaar";
 }
}
```

Then the "Foo" remote view is exposed under the `java:jboss/exported/` namespace under the following JNDI name scheme (which is similar to that mandated by EJB3.1 spec for `java:global/` namespace):[app-name]

`app-name/module-name/bean-name!bean-interface`

where,

`app-name` = the name of the .ear (without the .ear suffix) or the application name configured via application.xml deployment descriptor. If the application isn't packaged in a .ear then there will be **no** app-name part to the JNDI string.
`module-name` = the name of the .jar or .war (without the .jar/.war suffix) in which the bean is deployed or the module-name configured in web.xml/ejb-jar.xml of the deployment. The module name is mandatory part in the JNDI string.
`bean-name` = the name of the bean which by default is the simple name of the bean implementation class. Of course it can be overridden either by using the "name" attribute of the bean definining annotation (@Stateless(name="blah") in this case) or even the ejb-jar.xml deployment descriptor.
`bean-interface` = the fully qualified class name of the interface being exposed by the bean.

So in our example above, let's assume the bean is packaged in a myejbmodule.jar which is within a myapp.ear. So the JNDI name for the Foo remote view under the `java:jboss/exported/` namespace would be:

`java:jboss/exported/myapp/myejbmodule/FooBean!org.myapp.ejb.Foo`

That's where WildFly will **automatically** expose the remote views of the EJBs under the `java:jboss/exported/` namespace, **in addition to** the java:global/ java:app/ java:module/ namespaces mandated by the EJB 3.1 spec.

 Note that only the java:jboss/exported/ namespace is available to remote clients.

So the next logical question would be, are these remote views of EJBs accessible and invokable using the remote-naming project on the client application. The answer is *yes*! Let's quickly see the client code for invoking our `FooBean`. Again, let's just use "`peter`" and "`lois`" as username/pass for connecting to the remoting connector.

```
void doBeanLookup() {
  ...
  Properties jndiProps = new Properties();
  jndiProps.put(Context.INITIAL_CONTEXT_FACTORY,
"org.jboss.naming.remote.client.InitialContextFactory");
  jndiProps.put(Context.PROVIDER_URL,"http-remoting://localhost:8080");
  // username
  jndiProps.put(Context.SECURITY_PRINCIPAL, "peter");
  // password
  jndiProps.put(Context.SECURITY_CREDENTIALS, "lois");
  // This is an important property to set if you want to do EJB invocations via the
remote-naming project
  jndiProps.put("jboss.naming.client.ejb.context", true);
  // create a context passing these properties
  Context ctx = new InitialContext(jndiProps);
  // lookup the bean     Foo
  beanRemoteInterface = (Foo) ctx.lookup("myapp/myejbmodule/FooBean!org.myapp.ejb.Foo");
  String bar = beanRemoteInterface.sayBar();
  System.out.println("Remote Foo bean returned " + bar);
  ctx.close();
  // after this point the beanRemoteInterface is not longer valid!
}
```

As you can see, most of the code is similar to what we have been seeing so far for setting up a JNDI context backed by the remote-naming project. The only parts that change are:

1) An additional "`jboss.naming.client.ejb.context`" property that is added to the properties passed to the InitialContext constructor.
2) The JNDI name used for the lookup
3) And subsequently the invocation on the bean interface returned by the lookup.

Let's see what the "`jboss.naming.client.ejb.context`" does. In WildFly, remote access/invocations on EJBs is facilitated by the JBoss specific EJB client API, which is a project on its own https://github.com/jbossas/jboss-ejb-client. So no matter, what mechanism you use (remote-naming or core EJB client API), the invocations are ultimately routed through the EJB client API project. In this case too, the remote-naming internally uses EJB client API to handle EJB invocations. From a EJB client API project perspective, for successful communication with the server, the project expects a `EJBClientContext` backed by (atleast one) `EJBReceiver`(s). The `EJBReceiver` is responsible for handling the EJB invocations. One type of a `EJBReceiver` is a `RemotingConnectionEJBReceiver` which internally uses jboss-remoting project to communicate with the remote server to handle the EJB invocations. Such a `EJBReceiver` expects a connection backed by the jboss-remoting project. Of course to be able to connect to the server, such a `EJBReceiver` would have to know the server address, port, security credentials and other similar parameters. If you were using the core EJB client API, then you would have configured all these properties via the jboss-ejb-client.properties or via programatic API usage as explained here EJB invocations from a remote client using JNDI. But in the example above, we are using remote-naming project and are *not* directly interacting with the EJB client API project.

If you look closely at what's being passed, via the JNDI properties, to the remote-naming project and if you remember the details that we explained in a previous section about how the remote-naming project establishes a connection to the remote server, you'll realize that these properties are indeed the same as what the `RemotingConnectionEJBReceiver` would expect to be able to establish the connection to the server. Now this is where the "`jboss.naming.client.ejb.context`" property comes into picture. When this is set to true and passed to the InitialContext creation (either via jndi.properties or via the constructor of that class), the remote-naming project internally will do whatever is necessary to setup a `EJBClientContext`, containing a `RemotingConnectionEJBReceiver` which is created using the **same** remoting connection that is created by and being used by remote-naming project for its own JNDI communication usage. So effectively, the InitialContext creation via the remote-naming project has now internally triggered the creation of a `EJBClientContext` containing a `EJBReceiver` capable of handling the EJB invocations (remember, no remote EJB invocations are possible without the presence of a `EJBClientContext` containing a `EJBReceiver` which can handle the EJB).

So we now know the importance of the "`jboss.naming.client.ejb.context`" property and its usage. Let's move on the next part in that code, the JNDI name. Notice that we have used the JNDI name relative to the `java:jboss/exported/` namespace while doing the lookup. And since we know that the Foo view is exposed on that JNDI name, we cast the returned object back to the Foo interface. Remember that we earlier explained how each lookup via remote-naming triggers a server side communication and a marshalling/unmarshalling process. This applies for EJB views too. In fact, the remote-naming project has no clue (since that's not in the scope of that project to know) whether it's an EJB or some random object.

Once the unmarshalled object is returned (which actually is a proxy to the bean), the rest is straightforward, we just invoke on that returned object. Now since the remote-naming implementation has done the necessary setup for the EJBClientContext (due to the presence of "`jboss.naming.client.ejb.context`" property), the invocation on that proxy will internally use the `EJBClientContext` (the proxy is smart enough to do that) to interact with the server and return back the result. We won't go into the details of how the EJB client API handles the communication/invocation.

Long story short, using the remote-naming project for doing remote EJB invocations against WildFly is possible!

6.31.6 Why use the EJB client API approach then?

I can guess that some of you might already question why/when would one use the EJB client API style lookups as explained in the EJB invocations from a remote client using JNDI article instead of just using (what appears to be a simpler) remote-naming style lookups.

Before we answer that, let's understand a bit about the EJB client project. The EJB client project was implemented keeping in mind various optimizations and features that would be possible for handling remote invocations. One such optimization was to avoid doing unnecessary server side communication(s) which would typically involve network calls, marshalling/unmarshalling etc... The easiest place where this optimization can be applied, is to the EJB lookup. Consider the following code (let's ignore how the context is created):

```
ctx.lookup("foo/bar");
```

Now `foo/bar` JNDI name could potentially point to **any** type of object on the server side. The jndi name itself won't have the type/semantic information of the object bound to that name on the server side. If the context was setup using the remote-naming project (like we have seen earlier in our examples), then the only way for remote-naming to return an object for that lookup operation is to communicate with the server and marshal/unmarshal the object bound on the server side. And that's exactly what it does (remember, we explained this earlier).

The EJB client API project on the other hand optimizes this lookup. In order to do so, it expects the client application to let it know that a EJB is being looked up. It does this, by expecting the client application to use the JNDI name of the format "`ejb:`" namespace and also expecting the client application to setup the JNDI context by passing the "`org.jboss.ejb.client.naming`" value for the `Context.URL_PKG_PREFIXES` property.

Example:

```
final Properties jndiProperties = new Properties();
jndiProperties.put(Context.URL_PKG_PREFIXES, "org.jboss.ejb.client.naming");
// create the context
final Context context = new InitialContext(jndiProperties);

// lookup
Foo beanProxy = context.lookup("ejb:myapp/myejbmodule//FooBean!org.myapp.ejb.Foo");
String bar = beanProxy.sayBar();
```

More details about such code can be found here EJB invocations from a remote client using JNDI

When a client application looks up anything under the "`ejb:`" namespace, it is a clear indication (for the EJB client API project) to know that the client is looking up an EJB. That's where it steps in to do the necessary optimizations that might be applicable. So unlike, in the case of remote-naming project (which has no clue about the semantics of the object being looked up), the EJB client API project does **not** trigger a server side communication or a marshal/unmarshal process when you do lookup for a remote view of a stateless bean (it's important to note that we have specifically mentioned stateless bean here, we'll come to that later). Instead, the EJB client API just returns a java.lang.reflect.Proxy instance of the remote view type that's being looked up. This not just saves a network call, marshalling/unmarshalling step but it also means that you can create an EJB proxy even when the server isn't up yet. Later on, when the invocation on the proxy happens, the EJB client API *does* communicate with the server to carry out the invocation.

Is the lookup optimization applicable for all bean types?

In the previous section we (intentionally) mentioned that the lookup optimization by the EJB client API project happens for stateless beans. This kind of optimization is **not** possible for stateful beans because in case of stateful beans, a lookup is expected to create a session for that stateful bean and for session creation we do have to communicate with the server since the server is responsible for creating that session.

That's exactly why the EJB client API project expects the JNDI name lookup string for stateful beans to include the "`?stateful`" string at the end of the JNDI name:

```
context.lookup("ejb:myapp/myejbmodule//StatefulBean!org.myapp.ejb.Counter?stateful");
```

Notice the use of "`?stateful`" in that JNDI name. See EJB invocations from a remote client using JNDI for more details about such lookup.

The presence of "`?stateful`" in the JNDI name lookup string is a directive to the EJB client API to let it know that a stateful bean is being looked up and it's necessary to communicate with the server and create a session during that lookup.

So as you can see, we have managed to optimize certain operations by using the EJB client API for EJB lookup/invocation as against using the remote-naming project. There are other EJB client API implementation details (and probably more might be added) which are superior when it is used for remote EJB invocations in client applications as against remote-naming project which doesn't have the intelligence to carry out such optimizations for EJB invocations. *That's why the remote-naming project* ***for remote EJB invocations*** *is considered "*`deprecated`*"*. Note that if you want to use remote-naming for looking up and invoking on non-EJB remote objects then you are free to do so. In fact, that's why that project has been provided. You can even use the remote-naming project for EJB invocations (like we just saw), if you are fine with *not* wanting the optimizations that the EJB client API can do for you or if you have other restrictions that force you to use that project.

Restrictions for EJB's

If the remote-naming is used there are some restrictions as there is no full support of the ejb-client features.

- No loadbalancing, if the URL conatains multiple "remote://" servers there is no loadbalancing, the first available server will be used and only in case it is not longer available there will be a failover to the next available one.
- No cluster support. As a cluster needs to be defined in the jboss-ejb-client.properties this feature can not be used and there is no cluster node added
- No client side interceptor. The EJBContext.getCurrent() can not be used and it is not possible to add a client interceptor
- No UserTransaction support
- All proxies become invalid if .close() for the related Initalcontext is invoked, or the InitialContext is not longer referenced and gets garbage-collected. In this case the underlying EJBContext is destroyed and the conections are closed.
- It is not possible to use remote-naming if the client is an application deployed on another JBoss instance

6.32 Scoped EJB client contexts

6.32.1 Overview

WildFly 8 introduced the EJB client API for managing remote EJB invocations. The EJB client API works off EJBClientContext(s). An EJBClientContext can potentially contain any number of EJB receivers. An EJB receiver is a component which knows how to communicate with a server which is capable of handling the EJB invocation. Typically EJB remote applications can be classified into:

- A remote client which runs as a standalone Java application
- A remote client which runs within another WildFly 8 instance

Depending on the kind of remote client, from an EJB client API point of view, there can potentially be more than 1 EJBClientContext(s) within a JVM.

In case of standalone applications, typically a single EJBClientContext (backed by any number of EJB receivers) exists. However this isn't mandatory. Certain standalone applications can potentially have more than one EJBClientContext(s) and a EJB client context selector will be responsible for returning the appropriate context.

In case of remote clients which run within another WildFly 8 instance, each deployed application will have a corresponding EJB client context. Whenever that application invokes on another EJB, the corresponding EJB client context will be used for finding the right EJB receiver and letting it handle the invocation.

6.32.2 Potential shortcomings of a single EJB client context

In the Overview section we briefly looked at the different types of remote clients. Let's focus on the standalone remote clients (the ones that don't run within another WildFly 8 instance) for some of the next sections. Like mentioned earlier, typically a remote standalone client has just one EJB client context backed by any number of EJB receivers. Consider this example:

```
public class MyApplication {

    public static void main(String args[]) {

        final javax.naming.Context ctxOne = new javax.naming.InitialContext();
        final MyBeanInterface beanOne = ctxOne.lookup("ejb:app/module/distinct/bean!interface");
        beanOne.doSomething();
        ...
    }
}
```

Now, we have seen in this other chapter EJB invocations from a remote client using JNDI that the JNDI lookups are (typically) backed by jboss-ejb-client.properties file which is used to setup the EJB client context and the EJB receivers. Let's assume we have a jboss-ejb-client.properties with the relevant receivers configurations. These configurations include the security credentials that will be used to create a EJB receiver which connects to the AS7 server. Now when the above code is invoked, the EJB client API looks for the EJB client context to pick a EJB receiver, to pass on the EJB invocation request. Since we just have a single EJB client context, that context is used by the above code to invoke the bean.

Now let's consider a case where the user application wants to invoke on the bean more than once, but wants to connect to the WildFly 8 server using different security credentials. Let's take a look at the following code:

```
public class MyApplication {

    public static void main(String args[]) {

        // let's say we want to use "foo" security credential while connecting to the AS7 server
for invoking on this bean instance
        final javax.naming.Context ctxOne = new javax.naming.InitialContext();
        final MyBeanInterface beanOne = ctxOne.lookup("ejb:app/module/distinct/bean!interface");
        beanOne.doSomething();
        ...

        // let's say we want to use "bar" security credential while connecting to the AS7 server
for invoking on this bean instance
        final javax.naming.Context ctxTwo = new javax.naming.InitialContext();
        final MyBeanInterface beanTwo = ctxTwo.lookup("ejb:app/module/distinct/bean!interface");
        beanTwo.doSomething();
        ...

    }
}
```

So we have the same application, which wants to connect to the same server instance for invoking the EJB(s) hosted on that server, but wants to use two different credentials while connecting to the server. Remember, the client application has a single EJB client context which can have atmost 1 EJB recevier for each server instance. Which effectively means that the above code will end up using just one credential to connect to the server. So there was no easy way to have the above code working.

That was one of the use cases which prompted the https://issues.jboss.org/browse/EJBCLIENT-34 feature request. The proposal was to introduce a way, where you can have more control over the EJB client contexts and their association with JNDI contexts which are typically used for EJB invocations.

6.32.3 Scoped EJB client contexts

Developers familiar with earlier versions of JBoss AS would remember that for invoking an EJB, you would typically create a JNDI context passing it the PROVIDER_URL which would point to the target server. That way any invocation done on EJB proxies looked up using that JNDI context, would end up on that server. If we look back at the example above, we'll realize that, we are ultimately aiming for a similar functionality through https://issues.jboss.org/browse/EJBCLIENT-34. We want the user applications to have more control over which EJB receiver gets used for a specific invocation.

Before we introduced https://issues.jboss.org/browse/EJBCLIENT-34 feature, the EJB client context was typically scoped to the client application. As part of https://issues.jboss.org/browse/EJBCLIENT-34 we now allow the EJB client contexts to be scoped with the JNDI contexts. Consider the following example:

```
public class MyApplication {

    public static void main(String args[]) {

        // let's say we want to use "foo" security credential while connecting to the AS7 server
for invoking on this bean instance
        final Properties ejbClientContextPropsOne = getPropsForEJBClientContextOne():
        final javax.naming.Context ctxOne = new
javax.naming.InitialContext(ejbClientContextPropsOne);
        final MyBeanInterface beanOne = ctxOne.lookup("ejb:app/module/distinct/bean!interface");
        beanOne.doSomething();
        ...
        closeContext(ctxOne); // read on the entire article to understand more about closing
scoped EJB client contexts

        // let's say we want to use "bar" security credential while connecting to the AS7 server
for invoking on this bean instance
        final Properties ejbClientContextPropsTwo = getPropsForEJBClientContextTwo():
        final javax.naming.Context ctxTwo = new
javax.naming.InitialContext(ejbClientContextPropsTwo);
        final MyBeanInterface beanTwo = ctxTwo.lookup("ejb:app/module/distinct/bean!interface");
        beanTwo.doSomething();
        ...
        closeContext(ctxTwo); // read on the entire article to understand more about closing
scoped EJB client contexts
    }
}
```

Notice any difference between this code and the earlier one? We now create and pass EJB client context specific properties to the JNDI context. So what do the EJB client context properties look like? The properties are the same that you would pass through the jboss-ejb-client.properties file, except for one additional property which is required to scope the EJB client context to the JNDI context. The name of the property is:

```
org.jboss.ejb.client.scoped.context
```

which is expected to have a value true. This property lets the EJB client API know that it has to created a EJB client context (backed by EJB receiver(s)) and that created context is then scoped/visible to only that JNDI context which created it. Lookup and invocation on any EJB proxies looked up using this JNDI context will only know of the EJB client context associated with this JNDI context. This effectively means that the other JNDI contexts which the application uses to lookup and invoke on EJBs will *not* know about the other scoped EJB client contexts at all.

JNDI contexts which aren't scoped to a EJB client context (for example, not passing the org.jboss.ejb.client.scoped.context property) will fallback to the default behaviour of using the "current" EJB client context which typically is the one tied to the entire application.

This scoping of the EJB client context helps the user applications to have more control over which JNDI context "talks to" which server and connects to that server in "what way". This gives the user applications the flexibility that was associated with the JNP based JNDI invocations prior to WildFly 8 versions.

IMPORTANT: It is very important to remember that scoped EJB client contexts which are scoped to the JNDI contexts are NOT fire and forget kind of contexts. What that means is the application program which is using these contexts is solely responsible for managing their lifecycle and the application itself is responsible for closing the context at the right moment. After closing the context the proxies which are bound to this context are no longer valid and any invocation will throw an Exception. Not closing the context will end in resource problems as the underlying physical connection will stay open.

Read the rest of the sections in this article to understand more about the lifecycle management of such scoped contexts.

6.32.4 Lifecycle management of scoped EJB client contexts

Like you saw in the previous sections, in case of scoped EJB client contexts, the EJB client context is tied to the JNDI context. It's very important to understand how the lifecycle of the EJB client context works in such cases. Especially since any EJB client context is almost always backed by connections to the server. Not managing the EJB client context lifecycle correctly can lead to connection leaks in some cases.

When you create a scoped EJB client context, the EJB client context connects to the server(s) listed in the JNDI properties. An internal implementation detail of this logic includes the ability of the EJB client context to cache connections based on certain internal algorithm it uses. The algorithm itself isn't publicly documented (yet) since the chances of it changing or even removal shouldn't really affect the client application and instead it's supposed to be transparent to the client application.

The connections thus created for a EJB client context are kept open as long as the EJB client context is open. This allows the EJB client context to be usable for EJB invocations. The connections associated with the EJB client context are closed when the EJB client context itself is closed.

The connections that were manually added by the application to the EJB client context are **not** managed by the EJB client context. i.e. they won't be opened (obviously) nor closed by the EJB client API when the EJB client context is closed.

How to close EJB client contexts?

The answer to that is simple. Use the close() method on the appropriate EJB client context.

How to close scoped EJB client contexts?

The answer is the same, use the close() method on the EJB client context. But the real question is how do you get the relevant scoped EJB client context which is associated with a JNDI context. Before we get to that, it's important to understand how the ejb: JNDI namespace that's used for EJB lookups and how the JNDI context (typically the InitialContext that you see in the client code) are related. The JNDI API provided by Java language allows "URL context factory" to be registered in the JNDI framework (see this for details http://docs.oracle.com/javase/jndi/tutorial/provider/url/factory.html). Like that documentation states, the URL context factory can be used to resolve URL strings during JNDI lookup. That's what the ejb: prefix is when you do a remote EJB lookup. The ejb: URL string is backed by a URL context factory.

Internally, when a lookup happens for a ejb: URL string, a relevant javax.naming.Context is created for that ejb: lookup. Let's see some code for better understanding:

```
// JNDI context "A"
Context jndiCtx = new InitialContext(props);
// Now let's lookup a EJB
MyBean bean = jndiCtx.lookup("ejb:app/module/distinct/bean!interface");
```

So we first create a JNDI context and then use it to lookup an EJB. The bean lookup using the ejb: JNDI name, although, is just one statement, involves a few more things under the hood. What's actually happening when you lookup that string is that a separate javax.naming.Context gets created for the ejb: URL string. This new javax.naming.Context is then used to lookup the rest of the string in that JNDI name.

Let's break up that one line into multiple statements to understand better:

```
// Remember, the ejb: is backed by a URL context factory which returns a Context for the ejb:
URL (that's why it's called a context factory)
final Context ejbNamingContext = (Context) jndiCtx.lookup("ejb:");
// Use the returned EJB naming context to lookup the rest of the JNDI string for EJB
final MyBean bean = ejbNamingContext.lookup("app/module/distinct/bean!interface");
```

As you see above, we split up that single statement into a couple of statements for explaining the details better. So as you can see when the ejb: URL string is parsed in a JNDI name, it gets hold of a javax.naming.Context instance. This instance is different from the one which was used to do the lookup (jndiCtx in this example). This is an important detail to understand (for reasons explained later). Now this returned instance is used to lookup the rest of the JNDI string ("app/module/distinct/bean!interface"), which then returns the EJB proxy. Irrespective of whether the lookup is done in a single statement or multiple parts, the code works the same. i.e. an instance of javax.naming.Context gets created for the ejb: URL string.

So why am I explaining all this when the section is titled `"How to close scoped EJB client contexts"`? The reason is because client applications dealing with scoped EJB client contexts which are associated with a JNDI context would expect the following code to close the associated EJB client context, but will be surprised that it won't:

```
final Properties props = new Properties();
// mark it for scoped EJB client context
props.put("org.jboss.ejb.client.scoped.context","true");
// add other properties
props.put(....);
...
Context jndiCtx = new InitialContext(props);
try {
      final MyBean bean = jndiCtx.lookup("ejb:app/module/distinct/bean!interface");
      bean.doSomething();
} finally {
  jndiCtx.close();
}
```

Applications expect that the call to jndiCtx.close() will effectively close the EJB client context associated with the JNDI context. That doesn't happen because as explained previously, the javax.naming.Context backing the ejb: URL string is a different instance than the one the code is closing. The JNDI implementation in Java, only just closes the context on which the close was called. As a result, the other javax.naming.Context that backs the ejb: URL string is still not closed, which effectively means that the scoped EJB client context is not closed too which then ultimately means that the connection to the server(s) in the EJB client context are not closed too.

So now let's see how this can be done properly. We know that the ejb: URL string lookup returns us a javax.naming.Context. All we have to do is keep a reference to this instance and close it when we are done with the EJB invocations. So here's how it's going to look:

```
final Properties props = new Properties();
// mark it for scoped EJB client context
props.put("org.jboss.ejb.client.scoped.context","true");
// add other properties
props.put(....);
...
Context jndiCtx = new InitialContext(props);
Context ejbRootNamingContext = (Context) jndiCtx.lookup("ejb:");
try {
    final MyBean bean = ejbRootNamingContext.lookup("app/module/distinct/bean!interface"); //
the rest of the EJB jndi string
    bean.doSomething();
} finally {
    try {
        // close the EJB naming JNDI context
        ejbRootNamingContext.close();
    } catch (Throwable t) {
        // log and ignore
    }
    try {
        // also close our other JNDI context since we are done with it too
        jndiCtx.close();
    } catch (Throwable t) {
        // log and ignore
    }

}
```

As you see, we changed the code to first do a lookup on just the "ejb:" string to get hold of the EJB naming context and then used that ejbRootNamingContext instance to lookup the rest of the EJB JNDI name to get hold of the EJB proxy. Then when it was time to close the context, we closed the ejbRootNamingContext (as well as the other JNDI context). Closing the ejbRootNamingContext ensures that the scoped EJB client context associated with that JNDI context is closed too. Effectively, this closes the connection(s) to the server(s) within that EJB client context.

Can that code be simplified a bit?

If you are using that JNDI context only for EJB invocations, then yes you can get rid of some instances and code from the above code. You can change that code to:

```
final Properties props = new Properties();
// mark it for scoped EJB client context
props.put("org.jboss.ejb.client.scoped.context","true");
// add other properties
props.put(....);
...
Context ejbRootNamingContext = (Context) new InitialContext(props).lookup("ejb:");
try {
    final MyBean bean = ejbRootNamingContext.lookup("app/module/distinct/bean!interface"); //
the rest of the EJB jndi string
    bean.doSomething();
} finally {
    try {
        // close the EJB naming JNDI context
        ejbRootNamingContext.close();
    } catch (Throwable t) {
        // log and ignore
    }
}
```

Notice that we no longer hold a reference to 2 JNDI contexts and instead just keep track of the ejbRootNamingContext which is actually the root JNDI context for our "ejb:" URL string. Of course, this means that you can only use this context for EJB lookups or any other EJB related JNDI lookups. So it depends on your application and how it's coded.

Can't the scoped EJB client context be automatically closed by the EJB client API when the JNDI context is no longer in scope (i.e. on GC)?

That's one of the common questions that gets asked. No, the EJB client API can't take that decision. i.e. it cannot automatically go ahead and close the scoped EJB client context by itself when the associated JNDI context is eligible for GC. The reason is simple as illustrated by the following code:

```
void doEJBInvocation() {
    final MyBean bean = lookupEJB();
    bean.doSomething();
    bean.doSomeOtherThing();
    ... // do some other work
    bean.keepDoingSomething();
}

MyBean lookupEJB() {
    final Properties props = new Properties();
    // mark it for scoped EJB client context
    props.put("org.jboss.ejb.client.scoped.context","true");
    // add other properties
    props.put(....);
    ...
    Context ejbRootNamingContext = (Context) new InitialContext(props).lookup("ejb:");
    final MyBean bean = ejbRootNamingContext.lookup("app/module/distinct/bean!interface"); //
rest of the EJB jndi string
    return bean;
}
```

As you can see, the doEJBInvocation() method first calls a lookupEJB() method which does a lookup of the bean using a JNDI context and then returns the bean (proxy). The doEJBInvocation() then uses that returned proxy and keeps doing the invocations on the bean. As you might have noticed, the JNDI context that was used for lookup (i.e. the ejbRootNamingContext) is eligible for GC. If the EJB client API had closed the scoped EJB client context associated with that JNDI context, when that JNDI context was garbage collected, then the subsequent EJB invocations on the returned EJB (proxy) would start failing in doEJBInvocation() since the EJB client context is no longer available.

That's the reason why the EJB client API doesn't automatically close the EJB client context.

6.33 Spring applications development and migration guide

This document details the main points that need to be considered by Spring developers that wish to develop new applications or to migrate existing applications to be run into WildFly 8.

6.33.1 Dependencies and Modularity

WildFly 8 has a modular class loading strategy, different from previous versions of JBoss AS, which enforces a better class loading isolation between deployments and the application server itself. A detailed description can be found in the documentation dedicated to class loading in WildFly 8.

This reduces significantly the risk of running into a class loading conflict and allows applications to package their own dependencies if they choose to do so. This makes it easier for Spring applications that package their own dependencies - such as logging frameworks or persistence providers to run on WildFly 8.

At the same time, this does not mean that duplications and conflicts cannot exist on the classpath. Some module dependencies are implicit, depending on the type of deployment as shown here.

6.33.2 Persistence usage guide

Depending on the strategy being used, Spring applications can be:

- native Hibernate applications;
- JPA-based applications;
- native JDBC applications;

6.33.3 Native Spring/Hibernate applications

Applications that use the Hibernate API directly with Spring (i.e. through either one of LocalSessionFactoryBean or AnnotationSessionFactoryBean) may use a version of Hibernate 3 packaged inside the application. Hibernate 4 (which is provided through the 'org.hibernate' module of WildFly 8) is not supported by Spring 3.0 and Spring 3.1 (and may be supported by Spring 3.2 as described in SPR-8096), so adding this module as a dependency is not a solution.

6.33.4 based applications

Spring applications using JPA may choose between:

- using a server-deployed persistence unit;
- using a Spring-managed persistence unit.

Using server-deployed persistence units

Applications that use a server-deployed persistence unit must observe the typical Java EE rules in what concerns dependency management, i.e. the javax.persistence classes and persistence provider (Hibernate) are contained in modules which are added automatically by the application when the persistence unit is deployed.

In order to use the server-deployed persistence units from within Spring, either the persistence context or the persistence unit need to be registered in JNDI via web.xml as follows:

```
<persistence-context-ref>
    <persistence-context-ref-name>persistence/petclinic-em</persistence-unit-ref-name>
    <persistence-unit-name>petclinic</persistence-unit-name>
</persistence-context-ref>
```

or, respectively:

```
<persistence-unit-ref>
    <persistence-unit-ref-name>persistence/petclinic-emf</persistence-unit-ref-name>
    <persistence-unit-name>petclinic</persistence-unit-name>
</persistence-unit-ref>
```

When doing so, the persistence context or persistence unit are available to be looked up in JNDI, as follows:

```
<jee:jndi-lookup id="entityManager" jndi-name="java:comp/env/persistence/petclinic-em"
            expected-type="javax.persistence.EntityManager"/>
```

or

```
<jee:jndi-lookup id="entityManagerFactory" jndi-name="java:comp/env/persistence/petclinic-emf"
            expected-type="javax.persistence.EntityManagerFactory"/>
```

JNDI binding

JNDI binding via persistence.xml properties is not supported in WildFly 8.

Using Spring-managed persistence units

Spring applications running in WildFly 8 may also create persistence units on their own, using the LocalContainerEntityManagerFactoryBean. This is what these applications need to consider:

Placement of the persistence unit definitions

When the application server encounters a deployment that has a file named META-INF/persistence.xml (or, for that matter, WEB-INF/classes/META-INF/persistence.xml), it will attempt to create a persistence unit based on what is provided in the file. In most cases, such definition files are not compliant with the Java EE requirements, mostly because required elements such as the datasource of the persistence unit are supposed to be provided by the Spring context definitions, which will fail the deployment of the persistence unit, and consequently of the entire deployment.

Spring applications can easily avoid this type of conflict, by using a feature of the LocalContainerEntityManagerFactoryBean which is designed for this purpose. Persistence unit definition files can exist in other locations than META-INF/persistence.xml and the location can be indicated through the persistenceXmlLocation property of the factory bean class.

Assuming that the persistence unit is in the META-INF/jpa-persistence.xml, the corresponding definition can be:

```
<bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
       <property name="persistenceXmlLocation"
value="classpath*:META-INF/jpa-persistence.xml"/>
       <!-- other definitions -->
</bean>
```

Managing dependencies

Since the LocalContainerEntityManagerFactoryBean and the corresponding HibernateJpaVendorAdapter are based on Hibernate 3, it is required to use that version with the application. Therefore, the Hibernate 3 jars must be included in the deployment. At the same time, due the presence of @PersistenceUnit or @PersistenceContext annotations on the application classes, the application server will automatically add the 'org.hibernate' module as a dependency.

This can be avoided by instructing the server to exclude the module from the deployment's list of dependencies. In order to do so, include a META-INF/jboss-deployment-structure.xml or, for web applications, WEB-INF/jboss-deployment-structure.xml with the following content:

```
<jboss-deployment-structure xmlns="urn:jboss:deployment-structure:1.0">
  <deployment>
    <exclusions>
       <module name="org.hibernate"/>
    </exclusions>
  </deployment>
</jboss-deployment-structure>
```

6.34 Sharing sessions between wars in an ear

Undertow allows you to share sessions between wars in an ear, if it is explicitly configured to do so. Note that if you use this feature your applications may not be portable, as this is not a standard servlet feature.

In order to enable this you must include a `shared-session-config` element in the `jboss-all.xml` file in the META-INF directory of the ear:

```
<jboss umlns="urn:jboss:1.0">
    <shared-session-config xmlns="urn:jboss:shared-session-config:1.0">
        <session-config>
            <cookie-config>
                <path>/</path>
            </cookie-config>
        </session-config>
    </shared-session-config>
</jboss>
```

This element is used to configure the shared session manager that will be used by all wars in the ear. For full details of all the options provided by this file please see the schema at https://github.com/wildfly/wildfly/blob/master/undertow/src/main/resources/schema/shared-session-config_1_(
, however in general it mimics the options that are available in jboss-web.xml for configuring the session.

6.35 Webservices reference guide

The Web Services functionalities of WildFly are provided by the JBossWS project integration.

The latest project documentation is available here.

This section covers the most relevant topics for the JBossWS version available on WildFly 9.

- JAX-WS User Guide
- JAX-WS Tools
    - wsconsume
    - wsprovide
- Advanced User Guide
    - Predefined client and endpoint configurations
    - Authentication
    - Apache CXF integration
    - WS-Addressing
    - WS-Security
    - WS-Trust and STS
        - ActAs WS-Trust Scenario
        - OnBehalfOf WS-Trust Scenario
        - SAML Bearer Assertion Scenario
        - SAML Holder-Of-Key Assertion Scenario
    - WS-Reliable Messaging
    - SOAP over JMS
    - HTTP Proxy
    - WS-Discovery
    - WS-Policy
    - Published WSDL customization
- JBoss Modules and WS applications

6.35.1 WS User Guide

The Java API for XML-Based Web Services (JAX-WS / JSR-224) defines the mapping between WSDL and Java as well as the classes to be used for accessing webservices and publishing them. JBossWS implements the latest JAX-WS specification, hence users can reference it for any vendor agnostic webservice usage need. Below is a brief overview of the most basic functionalities.

Web Service Endpoints

JAX-WS simplifies the development model for a web service endpoint a great deal. In short, an endpoint implementation bean is annotated with JAX-WS annotations and deployed to the server. The server automatically generates and publishes the abstract contract (i.e. wsdl+schema) for client consumption. All marshalling/unmarshalling is delegated to JAXB.

Plain old Java Object (POJO)

Let's take a look at simple POJO endpoint implementation. All endpoint associated metadata is provided via JSR-181 annotations

```
@WebService
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class JSEBean01
{
   @WebMethod
   public String echo(String input)
   {
      ...
   }
}
```

The endpoint as a web application

A JAX-WS java service endpoint (JSE) is deployed as a web application. Here is a sample *web.xml* descriptor:

```
<web-app ...>
  <servlet>
    <servlet-name>TestService</servlet-name>
    <servlet-class>org.jboss.test.ws.jaxws.samples.jsr181pojo.JSEBean01</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>TestService</servlet-name>
    <url-pattern>/*</url-pattern>
  </servlet-mapping>
</web-app>
```

Packaging the endpoint

A JSR-181 java service endpoint (JSE) is packaged as a web application in a *war* file.

```
<war warfile="${build.dir}/libs/jbossws-samples-jsr181pojo.war"
webxml="${build.resources.dir}/samples/jsr181pojo/WEB-INF/web.xml">
  <classes dir="${build.dir}/classes">
    <include name="org/jboss/test/ws/samples/jsr181pojo/JSEBean01.class"/>
  </classes>
</war>
```

Note, only the endpoint implementation bean and web.xml are required.

Accessing the generated WSDL

A successfully deployed service endpoint will show up in the WildFly managent console. You can get the deployed endpoint wsdl address there too.

Note, it is also possible to generate the abstract contract off line using JBossWS tools. For details of that please see Bottom-Up (Java to WSDL).

EJB3 Stateless Session Bean (SLSB)

The JAX-WS programming model supports the same set of annotations on EJB3 stateless session beans as on POJO endpoints.

```
@Stateless
@Remote(EJB3RemoteInterface.class)
@RemoteBinding(jndiBinding = "/ejb3/EJB3EndpointInterface")

@WebService
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class EJB3Bean01 implements EJB3RemoteInterface
{
   @WebMethod
   public String echo(String input)
   {
      ...
   }
}
```

Above you see an EJB-3.0 stateless session bean that exposes one method both on the remote interface and as an endpoint operation.

Packaging the endpoint

A JSR-181 EJB service endpoint is packaged as an ordinary ejb deployment.

```
<jar jarfile="${build.dir}/libs/jbossws-samples-jsr181ejb.jar">
  <fileset dir="${build.dir}/classes">
    <include name="org/jboss/test/ws/samples/jsr181ejb/EJB3Bean01.class"/>
    <include name="org/jboss/test/ws/samples/jsr181ejb/EJB3RemoteInterface.class"/>
  </fileset>
</jar>
```

Accessing the generated WSDL

A successfully deployed service endpoint will show up in the WildFly managent console. You can get the deployed endpoint wsdl address there too.

Note, it is also possible to generate the abstract contract off line using JBossWS tools. For details of that please see Bottom-Up (Java to WSDL).

Endpoint Provider

JAX-WS services typically implement a native Java service endpoint interface (SEI), perhaps mapped from a WSDL port type, either directly or via the use of annotations.

Java SEIs provide a high level Java-centric abstraction that hides the details of converting between Java objects and their XML representations for use in XML-based messages. However, in some cases it is desirable for services to be able to operate at the XML message level. The Provider interface offers an alternative to SEIs and may be implemented by services wishing to work at the XML message level.

A Provider based service instances invoke method is called for each message received for the service.

```
@WebServiceProvider(wsdlLocation = "WEB-INF/wsdl/Provider.wsdl")
@ServiceMode(value = Service.Mode.PAYLOAD)
public class ProviderBeanPayload implements Provider<Source>
{
   public Source invoke(Source req)
   {
      // Access the entire request PAYLOAD and return the response PAYLOAD
   }
}
```

Note, `Service.Mode.PAYLOAD` is the default and does not have to be declared explicitly. You can also use `Service.Mode.MESSAGE` to access the entire SOAP message (i.e. with `MESSAGE` the Provider can also see SOAP Headers)

The abstract contract for a provider endpoint cannot be derived/generated automatically. Therefore it is necessary to specify the *wsdlLocation* with the `@WebServiceProvider` annotation.

Web Service Clients

Service

`Service` is an abstraction that represents a WSDL service. A WSDL service is a collection of related ports, each of which consists of a port type bound to a particular protocol and available at a particular endpoint address.

For most clients, you will start with a set of stubs generated from the WSDL. One of these will be the service, and you will create objects of that class in order to work with the service (see "static case" below).

Service Usage

Static case

Most clients will start with a WSDL file, and generate some stubs using JBossWS tools like *wsconsume*. This usually gives a mass of files, one of which is the top of the tree. This is the service implementation class.

The generated implementation class can be recognised as it will have two public constructors, one with no arguments and one with two arguments, representing the wsdl location (a `java.net.URL`) and the service name (a `javax.xml.namespace.QName`) respectively.

Usually you will use the no-argument constructor. In this case the WSDL location and service name are those found in the WSDL. These are set implicitly from the `@WebServiceClient` annotation that decorates the generated class.

The following code snippet shows the generated constructors from the generated class:

```
// Generated Service Class

@WebServiceClient(name="StockQuoteService", targetNamespace="http://example.com/stocks",
wsdlLocation="http://example.com/stocks.wsdl")
public class StockQuoteService extends javax.xml.ws.Service
{
   public StockQuoteService()
   {
      super(new URL("http://example.com/stocks.wsdl"), new QName("http://example.com/stocks",
"StockQuoteService"));
   }

   public StockQuoteService(String wsdlLocation, QName serviceName)
   {
      super(wsdlLocation, serviceName);
   }

   ...
}
```

Section Dynamic Proxy explains how to obtain a port from the service and how to invoke an operation on the port. If you need to work with the XML payload directly or with the XML representation of the entire SOAP message, have a look at `Dispatch`.

Dynamic case

In the dynamic case, when nothing is generated, a web service client uses `Service.create` to create Service instances, the following code illustrates this process.

```
URL wsdlLocation = new URL("http://example.org/my.wsdl");
QName serviceName = new QName("http://example.org/sample", "MyService");
Service service = Service.create(wsdlLocation, serviceName);
```

Handler Resolver

JAX-WS provides a flexible plug-in framework for message processing modules, known as handlers, that may be used to extend the capabilities of a JAX-WS runtime system. Handler Framework describes the handler framework in detail. A Service instance provides access to a `HandlerResolver` via a pair of `getHandlerResolver` / `setHandlerResolver` methods that may be used to configure a set of handlers on a per-service, per-port or per-protocol binding basis.

When a Service instance is used to create a proxy or a Dispatch instance then the handler resolver currently registered with the service is used to create the required handler chain. Subsequent changes to the handler resolver configured for a Service instance do not affect the handlers on previously created proxies, or Dispatch instances.

Executor

Service instances can be configured with a `java.util.concurrent.Executor`. The executor will then be used to invoke any asynchronous callbacks requested by the application. The `setExecutor` and `getExecutor` methods of `Service` can be used to modify and retrieve the executor configured for a service.

Dynamic Proxy

You can create an instance of a client proxy using one of `getPort` methods on the `Service`.

```
/**
 * The getPort method returns a proxy. A service client
 * uses this proxy to invoke operations on the target
 * service endpoint. The <code>serviceEndpointInterface</code>
 * specifies the service endpoint interface that is supported by
 * the created dynamic proxy instance.
 **/
public <T> T getPort(QName portName, Class<T> serviceEndpointInterface)
{
   ...
}

/**
 * The getPort method returns a proxy. The parameter
 * <code>serviceEndpointInterface</code> specifies the service
 * endpoint interface that is supported by the returned proxy.
 * In the implementation of this method, the JAX-WS
 * runtime system takes the responsibility of selecting a protocol
 * binding (and a port) and configuring the proxy accordingly.
 * The returned proxy should not be reconfigured by the client.
 *
 **/
public <T> T getPort(Class<T> serviceEndpointInterface)
{
   ...
}
```

The service endpoint interface (SEI) is usually generated using tools. For details see Top Down (WSDL to Java)

A generated static Service usually also offers typed methods to get ports. These methods also return dynamic proxies that implement the SEI.

```
@WebServiceClient(name = "TestEndpointService", targetNamespace = "http://org.jboss.ws/wsref",
   wsdlLocation = "http://localhost.localdomain:8080/jaxws-samples-webserviceref?wsdl")

public class TestEndpointService extends Service
{
    ...

    public TestEndpointService(URL wsdlLocation, QName serviceName) {
        super(wsdlLocation, serviceName);
    }

    @WebEndpoint(name = "TestEndpointPort")
    public TestEndpoint getTestEndpointPort()
    {
        return (TestEndpoint)super.getPort(TESTENDPOINTPORT, TestEndpoint.class);
    }
}
```

WebServiceRef

The `@WebServiceRef` annotation is used to declare a reference to a Web service. It follows the resource pattern exemplified by the `javax.annotation.Resource` annotation in JSR-250.

There are two uses to the WebServiceRef annotation:

1. To define a reference whose type is a generated service class. In this case, the type and value element will both refer to the generated service class type. Moreover, if the reference type can be inferred by the field/method declaration the annotation is applied to, the type and value elements MAY have the default value (Object.class, that is). If the type cannot be inferred, then at least the type element MUST be present with a non-default value.
2. To define a reference whose type is a SEI. In this case, the type element MAY be present with its default value if the type of the reference can be inferred from the annotated field/method declaration, but the value element MUST always be present and refer to a generated service class type (a subtype of javax.xml.ws.Service). The wsdlLocation element, if present, overrides theWSDL location information specified in the WebService annotation of the referenced generated service class.

   ```
   public class EJB3Client implements EJB3Remote
   {
      @WebServiceRef
      public TestEndpointService service4;

      @WebServiceRef
      public TestEndpoint port3;
   ```

Dispatch

XMLWeb Services use XML messages for communication between services and service clients. The higher level JAX-WS APIs are designed to hide the details of converting between Java method invocations and the corresponding XML messages, but in some cases operating at the XML message level is desirable. The Dispatch interface provides support for this mode of interaction.

`Dispatch` supports two usage modes, identified by the constants `javax.xml.ws.Service.Mode.MESSAGE` and `javax.xml.ws.Service.Mode.PAYLOAD` respectively:

Message In this mode, client applications work directly with protocol-specific message structures. E.g., when used with a SOAP protocol binding, a client application would work directly with a SOAP message.

Message Payload In this mode, client applications work with the payload of messages rather than the messages themselves. E.g., when used with a SOAP protocol binding, a client application would work with the contents of the SOAP Body rather than the SOAP message as a whole.

Dispatch is a low level API that requires clients to construct messages or message payloads as XML and requires an intimate knowledge of the desired message or payload structure. Dispatch is a generic class that supports input and output of messages or message payloads of any type.

```
Service service = Service.create(wsdlURL, serviceName);
Dispatch dispatch = service.createDispatch(portName, StreamSource.class, Mode.PAYLOAD);

String payload = "<ns1:ping xmlns:ns1='http://oneway.samples.jaxws.ws.test.jboss.org/'/>";
dispatch.invokeOneWay(new StreamSource(new StringReader(payload)));

payload = "<ns1:feedback xmlns:ns1='http://oneway.samples.jaxws.ws.test.jboss.org/'/>";
Source retObj = (Source)dispatch.invoke(new StreamSource(new StringReader(payload)));
```

Asynchronous Invocations

The `BindingProvider` interface represents a component that provides a protocol binding for use by clients, it is implemented by proxies and is extended by the `Dispatch` interface.

`BindingProvider` instances may provide asynchronous operation capabilities. When used, asynchronous operation invocations are decoupled from the `BindingProvider` instance at invocation time such that the response context is not updated when the operation completes. Instead a separate response context is made available using the `Response` interface.

```
public void testInvokeAsync() throws Exception
{
   URL wsdlURL = new URL("http://" + getServerHost() + ":8080/jaxws-samples-asynchronous?wsdl");
   QName serviceName = new QName(targetNS, "TestEndpointService");
   Service service = Service.create(wsdlURL, serviceName);
   TestEndpoint port = service.getPort(TestEndpoint.class);
   Response response = port.echoAsync("Async");
   // access future
   String retStr = (String) response.get();
   assertEquals("Async", retStr);
}
```

Oneway Invocations

`@Oneway` indicates that the given web method has only an input message and no output. Typically, a oneway method returns the thread of control to the calling application prior to executing the actual business method.

```
@WebService (name="PingEndpoint")
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class PingEndpointImpl
{
   private static String feedback;

   @WebMethod
   @Oneway
   publicvoid ping()
   {
      log.info("ping");
      feedback = "ok";
   }

   @WebMethod
   public String feedback()
   {
      log.info("feedback");
      return feedback;
   }
}
```

Timeout Configuration

There are two properties to configure the http connection timeout and client receive time out:

```
public void testConfigureTimeout() throws Exception
{
   //Set timeout until a connection is established
   ((BindingProvider)port).getRequestContext().put("javax.xml.ws.client.connectionTimeout",
"6000");

   //Set timeout until the response is received
   ((BindingProvider) port).getRequestContext().put("javax.xml.ws.client.receiveTimeout",
"1000");

   port.echo("testTimeout");
}
```

Common API

This sections describes concepts that apply equally to Web Service Endpoints and Web Service Clients.

Handler Framework

The handler framework is implemented by a JAX-WS protocol binding in both client and server side runtimes. Proxies, and Dispatch instances, known collectively as binding providers, each use protocol bindings to bind their abstract functionality to specific protocols.

Client and server-side handlers are organized into an ordered list known as a handler chain. The handlers within a handler chain are invoked each time a message is sent or received. Inbound messages are processed by handlers prior to binding provider processing. Outbound messages are processed by handlers after any binding provider processing.

Handlers are invoked with a message context that provides methods to access and modify inbound and outbound messages and to manage a set of properties. Message context properties may be used to facilitate communication between individual handlers and between handlers and client and service implementations. Different types of handlers are invoked with different types of message context.

Logical Handler

Handlers that only operate on message context properties and message payloads. Logical handlers are protocol agnostic and are unable to affect protocol specific parts of a message. Logical handlers are handlers that implement `javax.xml.ws.handler.LogicalHandler`.

Protocol Handler

Handlers that operate on message context properties and protocol specific messages. Protocol handlers are specific to a particular protocol and may access and change protocol specific aspects of a message. Protocol handlers are handlers that implement any interface derived from `javax.xml.ws.handler.Handler` except `javax.xml.ws.handler.LogicalHandler`.

Service endpoint handlers

On the service endpoint, handlers are defined using the `@HandlerChain` annotation.

```
@WebService
@HandlerChain(file = "jaxws-server-source-handlers.xml")
public class SOAPEndpointSourceImpl
{
   ...
}
```

The location of the handler chain file supports 2 formats

1. An absolute java.net.URL in externalForm. (ex: http://myhandlers.foo.com/handlerfile1.xml)

2. A relative path from the source file or class file. (ex: bar/handlerfile1.xml)

Service client handlers

On the client side, handler can be configured using the `@HandlerChain` annotation on the SEI or dynamically using the API.

```
Service service = Service.create(wsdlURL, serviceName);
Endpoint port = (Endpoint)service.getPort(Endpoint.class);

BindingProvider bindingProvider = (BindingProvider)port;
List<Handler> handlerChain = new ArrayList<Handler>();
handlerChain.add(new LogHandler());
handlerChain.add(new AuthorizationHandler());
handlerChain.add(new RoutingHandler());
bindingProvider.getBinding().setHandlerChain(handlerChain); // important!
```

Message Context

`MessageContext` is the super interface for all JAX-WS message contexts. It extends `Map<String,Object>` with additional methods and constants to manage a set of properties that enable handlers in a handler chain to share processing related state. For example, a handler may use the put method to insert a property in the message context that one or more other handlers in the handler chain may subsequently obtain via the get method.

Properties are scoped as either APPLICATION or HANDLER. All properties are available to all handlers for an instance of an MEP on a particular endpoint. E.g., if a logical handler puts a property in the message context, that property will also be available to any protocol handlers in the chain during the execution of an MEP instance. APPLICATION scoped properties are also made available to client applications (see section 4.2.1) and service endpoint implementations. The defaultscope for a property is HANDLER.

Logical Message Context

Logical Handlers are passed a message context of type `LogicalMessageContext` when invoked. `LogicalMessageContext` extends `MessageContext` with methods to obtain and modify the message payload, it does not provide access to the protocol specific aspects of amessage. A protocol binding defines what component of a message are available via a logical message context. The SOAP binding defines that a logical handler deployed in a SOAP binding can access the contents of the SOAP body but not the SOAP headers whereas the XML/HTTP binding defines that a logical handler can access the entire XML payload of a message.

SOAP Message Context

SOAP handlers are passed a `SOAPMessageContext` when invoked. `SOAPMessageContext` extends `MessageContext` with methods to obtain and modify the SOAP message payload.

Fault Handling

An implementation may thow a `SOAPFaultException`

```
public void throwSoapFaultException()
{
   SOAPFactory factory = SOAPFactory.newInstance();
   SOAPFault fault = factory.createFault("this is a fault string!", new QName("http://foo",
"FooCode"));
   fault.setFaultActor("mr.actor");
   fault.addDetail().addChildElement("test");
   thrownew SOAPFaultException(fault);
}
```

or an application specific user exception

```
public void throwApplicationException() throws UserException
{
   thrownew UserException("validation", 123, "Some validation error");
}
```

In case of the latter, JBossWS generates the required fault wrapper beans at runtime if they are not part of the deployment

WS Annotations

For details, see JSR-224 - Java API for XML-Based Web Services (JAX-WS) 2.2

javax.xml.ws.ServiceMode

The `ServiceMode` annotation is used to specify the mode for a provider class, i.e. whether a provider wants to have access to protocol message payloads (e.g. a SOAP body) or the entire protocol messages (e.g. a SOAP envelope).

javax.xml.ws.WebFault

The `WebFault` annotation is used when mapping WSDL faults to Java exceptions, see section 2.5. It is used to capture the name of the fault element used when marshalling the JAXB type generated from the global element referenced by the WSDL fault message. It can also be used to customize the mapping of service specific exceptions to WSDL faults.

javax.xml.ws.RequestWrapper

The `RequestWrapper` annotation is applied to the methods of an SEI. It is used to capture the JAXB generated request wrapper bean and the element name and namespace for marshalling / unmarshalling the bean. The default value of localName element is the operationName as defined in `WebMethod` annotation and the default value for the targetNamespace element is the target namespace of the SEI.When starting from Java, this annotation is used to resolve overloading conflicts in document literal mode. Only the className element is required in this case.

javax.xml.ws.ResponseWrapper

The `ResponseWrapper` annotation is applied to the methods of an SEI. It is used to capture the JAXB generated response wrapper bean and the element name and namespace for marshalling / unmarshalling the bean. The default value of the localName element is the operationName as defined in the `WebMethod` appended with ”Response” and the default value of the targetNamespace element is the target namespace of the SEI. When starting from Java, this annotation is used to resolve overloading conflicts in document literal mode. Only the className element is required in this case.

javax.xml.ws.WebServiceClient

The `WebServiceClient` annotation is specified on a generated service class (see 2.7). It is used to associate a class with a specific Web service, identify by a URL to a WSDL document and the qualified name of a wsdl:service element.

javax.xml.ws.WebEndpoint

The `WebEndpoint` annotation is specified on the getPortName() methods of a generated service class (see 2.7). It is used to associate a get method with a specific wsdl:port, identified by its local name (a NCName).

javax.xml.ws.WebServiceProvider

The `WebServiceProvider` annotation is specified on classes that implement a strongly typed `javax.xml.ws.Provider`. It is used to declare that a class that satisfies the requirements for a provider (see 5.1) does indeed define a Web service endpoint, much like the `WebService` annotation does for SEI-based endpoints.

The `WebServiceProvider` and `WebService` annotations are mutually exclusive.

javax.xml.ws.BindingType

The `BindingType` annotation is applied to an endpoint implementation class. It specifies the binding to use when publishing an endpoint of this type.

The default binding for an endpoint is the SOAP 1.1/HTTP one.

javax.xml.ws.WebServiceRef

The `WebServiceRef` annotation is used to declare a reference to a Web service. It follows the resource pattern exemplified by the `javax.annotation.Resource` annotation in JSR-250 [JBWS:32]. The `WebServiceRef` annotation is required to be honored when running on the Java EE 5 platform, where it is subject to the common resource injection rules described by the platform specification [JBWS:33].

javax.xml.ws.WebServiceRefs

The `WebServiceRefs` annotation is used to declare multiple references to Web services on a single class. It is necessary to work around the limition against specifying repeated annotations of the same type on any given class, which prevents listing multiple `javax.ws.WebServiceRef` annotations one after the other. This annotation follows the resource pattern exemplified by the `javax.annotation.Resources` annotation in JSR-250.

Since no name and type can be inferred in this case, each `WebServiceRef` annotation inside a WebServiceRefs MUST contain name and type elements with non-default values. The `WebServiceRef` annotation is required to be honored when running on the Java EE 5 platform, where it is subject to the common resource injection rules described by the platform specification.

javax.xml.ws.Action

The `Action` annotation is applied to the methods of a SEI. It used to generate the wsa:Action on wsdl:input and wsdl:output of each wsdl:operation mapped from the annotated methods.

javax.xml.ws.FaultAction

The `FaultAction` annotation is used within the `Action` annotation to generate the wsa:Action element on the wsdl:fault element of each wsdl:operation mapped from the annotated methods.

181 Annotations

JSR-181 defines the syntax and semantics of Java Web Service (JWS) metadata and default values.

For details, see JSR 181 - Web Services Metadata for the Java Platform.

javax.jws.WebService

Marks a Java class as implementing a Web Service, or a Java interface as defining a Web Service interface.

javax.jws.WebMethod

Customizes a method that is exposed as a Web Service operation.

javax.jws.OneWay

Indicates that the given web method has only an input message and no output. Typically, a oneway method returns the thread of control to the calling application prior to executing the actual business method. A JSR-181 processor is REQUIRED to report an error if an operation marked `@Oneway` has a return value, declares any checked exceptions or has any INOUT or OUT parameters.

javax.jws.WebParam

Customizes the mapping of an individual parameter to a Web Service message part and XML element.

javax.jws.WebResult

Customizes the mapping of the return value to a WSDL part and XML element.

javax.jws.SOAPBinding

Specifies the mapping of the Web Service onto the SOAP message protocol.

The `SOAPBinding` annotation has a target of `TYPE` and `METHOD`. The annotation may be placed on a method if and only if the `SOAPBinding.style` is `DOCUMENT`. Implementations MUST report an error if the `SOAPBinding` annotation is placed on a method with a `SOAPBinding.style` of `RPC`. Methods that do not have a `SOAPBinding` annotation accept the `SOAPBinding` behavior defined on the type.

javax.jws.HandlerChain

The `@HandlerChain` annotation associates the Web Service with an externally defined handler chain.

It is an error to combine this annotation with the `@SOAPMessageHandlers` annotation.

The `@HandlerChain` annotation MAY be present on the endpoint interface and service implementation bean. The service implementation bean's `@HandlerChain` is used if `@HandlerChain` is present on both.

The `@HandlerChain` annotation MAY be specified on the type only. The annotation target includes `METHOD` and `FIELD` for use by JAX-WS-2.x.

6.35.2 WS Tools

The JAX-WS tools provided by JBossWS can be used in a variety of ways. First we will look at server-side development strategies, and then proceed to the client.

Server side

When developing a Web Service Endpoint (the server-side) you have the option of starting from Java (*bottom-up development*), or from the abstact contract (WSDL) that defines your service (*top-down development*). If this is a new service (no existing contract), the bottom-up approach is the fastest route; you only need to add a few annotations to your classes to get a service up and running. However, if you are developing a service with an already defined contract, it is far simpler to use the top-down approach, since the provided tool will generate the annotated code for you.

Bottom-up use cases:

- Exposing an already existing EJB3 bean as a Web Service
- Providing a new service, and you want the contract to be generated for you

Top-down use cases:

- Replacing the implementation of an existing Web Service, and you can't break compatibility with older clients
- Exposing a service that conforms to a contract specified by a third party (e.g. a vender that calls you back using an already defined protocol).
- Creating a service that adheres to the XML Schema and WSDL you developed by hand up front

The following JAX-WS command line tools are included in JBossWS:

Command	Description
wsprovide	Generates JAX-WS portable artifacts, and provides the abstract contract. Used for bottom-up development.
wsconsume	Consumes the abstract contract (WSDL and Schema files), and produces artifacts for both a server and client. Used for top-down and client development

Bottom-Up (Using wsprovide)

The bottom-up strategy involves developing the Java code for your service, and then annotating it using JAX-WS annotations. These annotations can be used to customize the contract that is generated for your service. For example, you can change the operation name to map to anything you like. However, all of the annotations have sensible defaults, so only the @WebService annotation is required.

This can be as simple as creating a single class:

```
package echo;

@javax.jws.WebService
public class Echo
{
   public String echo(String input)
   {
      return input;
   }
}
```

A JSE or EJB3 deployment can be built using this class, and it is the only Java code needed to deploy on JBossWS. The WSDL, and all other Java artifacts called "wrapper classes" will be generated for you at deploy time. This actually goes beyond the JAX-WS specification, which requires that wrapper classes be generated using an offline tool. The reason for this requirement is purely a vender implementation problem, and since we do not believe in burdening a developer with a bunch of additional steps, we generate these as well. However, if you want your deployment to be portable to other application servers, you will unfortunately need to use a tool and add the generated classes to your deployment.

This is the primary purpose of the *wsprovide* tool, to generate portable JAX-WS artifacts. Additionally, it can be used to "provide" the abstract contract (WSDL file) for your service. This can be obtained by invoking *wsprovide* using the "-w" option:

```
$ javac -d . -classpath jboss-jaxws.jar Echo.java
$ wsprovide -w echo.Echo
Generating WSDL:
EchoService.wsdl
Writing Classes:
echo/jaxws/Echo.class
echo/jaxws/EchoResponse.class
```

Inspecting the WSDL reveals a service called *EchoService*:

```
<service name='EchoService'>
 <port binding='tns:EchoBinding' name='EchoPort'>
  <soap:address location='REPLACE_WITH_ACTUAL_URL'/>
 </port>
</service>
```

As expected, this service defines one operation, "*echo*":

```
<portType name='Echo'>
 <operation name='echo' parameterOrder='echo'>
  <input message='tns:Echo_echo'/>
  <output message='tns:Echo_echoResponse'/>
 </operation>
</portType>
```

Remember that when deploying on JBossWS you do not need to run this tool. You only need it for generating portable artifacts and/or the abstract contract for your service.

Let's create a POJO endpoint for deployment on WildFly. A simple *web.xml* needs to be created:

```
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
  version="2.4">

  <servlet>
    <servlet-name>Echo</servlet-name>
    <servlet-class>echo.Echo</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>Echo</servlet-name>
    <url-pattern>/Echo</url-pattern>
  </servlet-mapping>
</web-app>
```

The *web.xml* and the single class can now be used to create a war:

```
$ mkdir -p WEB-INF/classes
$ cp -rp echo WEB-INF/classes/
$ cp web.xml WEB-INF
$ jar cvf echo.war WEB-INF
added manifest
adding: WEB-INF/(in = 0) (out= 0)(stored 0%)
adding: WEB-INF/classes/(in = 0) (out= 0)(stored 0%)
adding: WEB-INF/classes/echo/(in = 0) (out= 0)(stored 0%)
adding: WEB-INF/classes/echo/Echo.class(in = 340) (out= 247)(deflated 27%)
adding: WEB-INF/web.xml(in = 576) (out= 271)(deflated 52%)
```

The war can then be deployed to the JBoss Application Server.The war can then be deployed to the JBoss Application Server; this will internally invoke wsprovide, which will generate the WSDL. If deployment was successful, and you are using the default settings, it should be available in the server management console.

For a portable JAX-WS deployment, the wrapper classes generated earlier could be added to the deployment.

Down (Using wsconsume)

The top-down development strategy begins with the abstract contract for the service, which includes the WSDL file and zero or more schema files. The *wsconsume* tool is then used to consume this contract, and produce annotated Java classes (and optionally sources) that define it.

wsconsume may have problems with symlinks on Unix systems

Using the WSDL file from the bottom-up example, a new Java implementation that adheres to this service can be generated. The "-k" option is passed to *wsconsume* to preserve the Java source files that are generated, instead of providing just classes:

```
$ wsconsume -k EchoService.wsdl
echo/Echo.java
echo/EchoResponse.java
echo/EchoService.java
echo/Echo_Type.java
echo/ObjectFactory.java
echo/package-info.java
echo/Echo.java
echo/EchoResponse.java
echo/EchoService.java
echo/Echo_Type.java
echo/ObjectFactory.java
echo/package-info.java
```

The following table shows the purpose of each generated file:

File	Purpose
Echo.java	Service Endpoint Interface
Echo_Type.java	Wrapper bean for request message
EchoResponse.java	Wrapper bean for response message
ObjectFactory.java	JAXB XML Registry
package-info.java	Holder for JAXB package annotations
EchoService.java	Used only by JAX-WS clients

Examining the Service Endpoint Interface reveals annotations that are more explicit than in the class written by hand in the bottom-up example, however, these evaluate to the same contract:

```
@WebService(name = "Echo", targetNamespace = "http://echo/")
public interface Echo {
    @WebMethod
    @WebResult(targetNamespace = "")
    @RequestWrapper(localName = "echo", targetNamespace = "http://echo/", className =
"echo.Echo_Type")
    @ResponseWrapper(localName = "echoResponse", targetNamespace = "http://echo/", className =
"echo.EchoResponse")
    public String echo(
        @WebParam(name = "arg0", targetNamespace = "")
        String arg0);

}
```

The only missing piece (besides for packaging) is the implementation class, which can now be written, using the above interface.

```
package echo;

@javax.jws.WebService(endpointInterface="echo.Echo")
public class EchoImpl implements Echo
{
   public String echo(String arg0)
   {
      return arg0;
   }
}
```

Client Side

Before going to detail on the client-side it is important to understand the decoupling concept that is central to Web Services. Web Services are not the best fit for internal RPC, even though they can be used in this way. There are much better technologies for this (CORBA, and RMI for example). Web Services were designed specifically for interoperable coarse-grained correspondence. There is no expectation or guarantee that any party participating in a Web Service interaction will be at any particular location, running on any particular OS, or written in any particular programming language. So because of this, it is important to clearly separate client and server implementations. The only thing they should have in common is the abstract contract definition. If, for whatever reason, your software does not adhere to this principal, then you should not be using Web Services. For the above reasons, the ***recommended methodology for developing a client is*** to follow ***the top-down approach***, even if the client is running on the same server.

Let's repeat the process of the top-down section, although using the deployed WSDL, instead of the one generated offline by *wsprovide*. The reason why we do this is just to get the right value for soap:address. This value must be computed at deploy time, since it is based on container configuration specifics. You could of course edit the WSDL file yourself, although you need to ensure that the path is correct.

Offline version:

```
<service name='EchoService'>
  <port binding='tns:EchoBinding' name='EchoPort'>
   <soap:address location='REPLACE_WITH_ACTUAL_URL'/>
  </port>
</service>
```

Online version:

```
<service name="EchoService">
  <port binding="tns:EchoBinding" name="EchoPort">
    <soap:address location="http://localhost.localdomain:8080/echo/Echo"/>
  </port>
</service>
```

Using the online deployed version with *wsconsume*:

```
$ wsconsume -k http://localhost:8080/echo/Echo?wsdl
echo/Echo.java
echo/EchoResponse.java
echo/EchoService.java
echo/Echo_Type.java
echo/ObjectFactory.java
echo/package-info.java
echo/Echo.java
echo/EchoResponse.java
echo/EchoService.java
echo/Echo_Type.java
echo/ObjectFactory.java
echo/package-info.java
```

The one class that was not examined in the top-down section, was `EchoService.java`. Notice how it stores the location the WSDL was obtained from.

```
@WebServiceClient(name = "EchoService", targetNamespace = "http://echo/", wsdlLocation =
"http://localhost:8080/echo/Echo?wsdl")
public class EchoService extends Service
{
    private final static URL ECHOSERVICE_WSDL_LOCATION;

    static {
        URL url = null;
        try
        {
           url = new URL("http://localhost:8080/echo/Echo?wsdl");
        }
        catch (MalformedURLException e)
        {
           e.printStackTrace();
        }
        ECHOSERVICE_WSDL_LOCATION = url;
    }

    public EchoService(URL wsdlLocation, QName serviceName)
    {
         super(wsdlLocation, serviceName);
    }

    public EchoService()
    {
         super(ECHOSERVICE_WSDL_LOCATION, new QName("http://echo/", "EchoService"));
    }

    @WebEndpoint(name = "EchoPort")
    public Echo getEchoPort()
    {
         return (Echo)super.getPort(new QName("http://echo/", "EchoPort"), Echo.class);
    }
}
```

As you can see, this generated class extends the main client entry point in JAX-WS, `javax.xml.ws.Service`. While you can use `Service` directly, this is far simpler since it provides the configuration info for you. The only method we really care about is the `getEchoPort()` method, which returns an instance of our Service Endpoint Interface. Any WS operation can then be called by just invoking a method on the returned interface.

It's not recommended to refer to a remote WSDL URL in a production application. This causes network I/O every time you instantiate the Service Object. Instead, use the tool on a saved local copy, or use the URL version of the constructor to provide a new WSDL location.

All that is left to do, is write and compile the client:

```
import echo.*;

public class EchoClient
{
   public static void main(String args[])
   {
      if (args.length != 1)
      {
          System.err.println("usage: EchoClient <message>");
          System.exit(1);
      }

      EchoService service = new EchoService();
      Echo echo = service.getEchoPort();
      System.out.println("Server said: " + echo.echo(args0));
   }
}
```

It is easy to change the endpoint address of your operation at runtime, setting the *ENDPOINT_ADDRESS_PROPERTY* as shown below:

```
EchoService service = new EchoService();
      Echo echo = service.getEchoPort();

      /* Set NEW Endpoint Location */
      String endpointURL = "http://NEW_ENDPOINT_URL";
      BindingProvider bp = (BindingProvider)echo;
      bp.getRequestContext().put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, endpointURL);

      System.out.println("Server said: " + echo.echo(args0));
```

wsconsume

wsconsume is a command line tool and ant task that "consumes" the abstract contract (WSDL file) and produces portable JAX-WS service and client artifacts.

Command Line Tool

The command line tool has the following usage:

```
usage: wsconsume [options] <wsdl-url>
options:
  -h, --help                  Show this help message
  -b, --binding=<file>        One or more JAX-WS or JAXB binding files
  -k, --keep                  Keep/Generate Java source
  -c  --catalog=<file>        Oasis XML Catalog file for entity resolution
  -j  --clientjar=<name>      Create a jar file of the generated artifacts for calling the
webservice
  -p  --package=<name>        The target package for generated source
  -w  --wsdlLocation=<loc>    Value to use for @WebServiceClient.wsdlLocation
  -o, --output=<directory>    The directory to put generated artifacts
  -s, --source=<directory>    The directory to put Java source
  -t, --target=<2.0|2.1|2.2>  The JAX-WS specification target
  -q, --quiet                 Be somewhat more quiet
  -v, --verbose               Show full exception stack traces
  -l, --load-consumer         Load the consumer and exit (debug utility)
  -e, --extension             Enable SOAP 1.2 binding extension
  -a, --additionalHeaders     Enables processing of implicit SOAP headers
  -n, --nocompile             Do not compile generated sources
```

The wsdlLocation is used when creating the Service to be used by clients and will be added to the @WebServiceClient annotation, for an endpoint implementation based on the generated service endpoint interface you will need to manually add the wsdlLocation to the @WebService annotation on your web service implementation and not the service endpoint interface.

Examples

Generate artifacts in Java class form only:

```
wsconsume Example.wsdl
```

Generate source and class files:

```
wsconsume -k Example.wsdl
```

Generate source and class files in a custom directory:

```
wsconsume -k -o custom Example.wsdl
```

Generate source and class files in the org.foo package:

```
wsconsume -k -p org.foo Example.wsdl
```

Generate source and class files using multiple binding files:

```
wsconsume -k -b wsdl-binding.xml -b schema1-binding.xml -b schema2-binding.xml
```

Maven Plugin

The wsconsume tools is included in the **org.jboss.ws.plugins:jaxws-tools-maven-plugin** plugin. The plugin has two goals for running the tool, *wsconsume* and *wsconsume-test*, which basically do the same during different maven build phases (the former triggers the sources generation during *generate-sources* phase, the latter during the *generate-test-sources* one).

The *wsconsume* plugin has the following parameters:

Attribute	Description	Default
bindingFiles	JAXWS or JAXB binding file	true
classpathElements	Each classpathElement provides a library file to be added to classpath	${project.compileClasspath or ${project.testClasspathEle
catalog	Oasis XML Catalog file for entity resolution	none
targetPackage	The target Java package for generated code.	generated
bindingFiles	One or more JAX-WS or JAXB binding file	none
wsdlLocation	Value to use for @WebServiceClient.wsdlLocation	generated
outputDirectory	The output directory for generated artifacts.	${project.build.outputDirect or ${project.build.testOutputD
sourceDirectory	The output directory for Java source.	${project.build.directory}/ws
verbose	Enables more informational output about command progress.	false
wsdls	The WSDL files or URLs to consume	n/a
extension	Enable SOAP 1.2 binding extension.	false
encoding	The charset encoding to use for generated sources.	${project.build.sourceEnco
argLine	An optional additional argline to be used when running in fork mode; can be used to set endorse dir, enable debugging, etc. ***Example*** `<argLine>-Djava.endorsed.dirs=...</argLine>`	none
fork	Whether or not to run the generation task in a separate VM.	false
target	A preference for the JAX-WS specification target	Depends on the underlying endorsed dirs if any

Examples

You can use *wsconsume* in your own project build simply referencing the *jaxws-tools-maven-plugin* in the configured plugins in your pom.xml file.

The following example makes the plugin consume the test.wsdl file and generate SEI and wrappers' java sources. The generated sources are then compiled together with the other project classes.

```
<build>
  <plugins>
    <plugin>
      <groupId>org.jboss.ws.plugins</groupId>
      <artifactId>jaxws-tools-maven-plugin</artifactId>
      <version>1.2.0.Beta1</version>
      <configuration>
        <wsdls>
          <wsdl>${basedir}/test.wsdl</wsdl>
        </wsdls>
      </configuration>
      <executions>
        <execution>
          <goals>
            <goal>wsconsume</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

You can also specify multiple wsdl files, as well as force the target package, enable SOAP 1.2 binding and turn the tool's verbose mode on:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.jboss.ws.plugins</groupId>
      <artifactId>jaxws-tools-maven-plugin</artifactId>
      <version>1.2.0.Beta1</version>
      <configuration>
       <wsdls>
        <wsdl>${basedir}/test.wsdl</wsdl>
        <wsdl>${basedir}/test2.wsdl</wsdl>
       </wsdls>
       <targetPackage>foo.bar</targetPackage>
       <extension>true</extension>
       <verbose>true</verbose>
      </configuration>
      <executions>
        <execution>
          <goals>
            <goal>wsconsume</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

Finally, if the wsconsume invocation is required for consuming a wsdl to be used in your testsuite only, you might want to use the *wsconsume-test* goal as follows:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.jboss.ws.plugins</groupId>
      <artifactId>jaxws-tools-maven-plugin</artifactId>
      <version>1.2.0.Beta1</version>
      <configuration>
        <wsdls>
          <wsdl>${basedir}/test.wsdl</wsdl>
        </wsdls>
      </configuration>
      <executions>
        <execution>
          <goals>
            <goal>wsconsume-test</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

Plugin stack dependencyThe plugin itself does not have an explicit dependency to a JBossWS stack, as it's meant for being used with implementations of any supported version of the *JBossWS SPI*. So the user is expected to set a dependency in his own `pom.xml` to the desired *JBossWS* stack version. The plugin will rely on the that for using the proper tooling.

```
<dependencies>
  <dependency>
    <groupId>org.jboss.ws.cxf</groupId>
    <artifactId>jbossws-cxf-client</artifactId>
    <version>4.0.0.GA</version>
  </dependency>
</dependencies>
```

Be careful when using this plugin with the Maven War Plugin as that include any project dependency into the generated application war archive. You might want to set `<scope>provided</scope>` for the *JBossWS* stack dependency to avoid that.

Up to version 1.1.2.Final, the *artifactId* of the plugin was **maven-jaxws-tools-plugin**.

Ant Task

The *wsconsume* Ant task (*org.jboss.ws.tools.ant.WSConsumeTask*) has the following attributes:

Attribute	Description	Default
fork	Whether or not to run the generation task in a separate VM.	true
keep	Keep/Enable Java source code generation.	false
catalog	Oasis XML Catalog file for entity resolution	none
package	The target Java package for generated code.	generated
binding	A JAX-WS or JAXB binding file	none
wsdlLocation	Value to use for @WebServiceClient.wsdlLocation	generated
encoding	The charset encoding to use for generated sources	n/a
destdir	The output directory for generated artifacts.	"output"
sourcedestdir	The output directory for Java source.	value of destdir
target	The JAX-WS specification target. Allowed values are 2.0, 2.1 and 2.2	
verbose	Enables more informational output about command progress.	false
wsdl	The WSDL file or URL	n/a
extension	Enable SOAP 1.2 binding extension.	false
additionalHeaders	Enables processing of implicit SOAP headers	false


Users also need to put streamBuffer.jar and stax-ex.jar to the classpath of the ant task to generate the appropriate artefacts.


The wsdlLocation is used when creating the Service to be used by clients and will be added to the @WebServiceClient annotation, for an endpoint implementation based on the generated service endpoint interface you will need to manually add the wsdlLocation to the @WebService annotation on your web service implementation and not the service endpoint interface.

Also, the following nested elements are supported:

Element	Description	Default
binding	A JAXWS or JAXB binding file	none
jvmarg	Allows setting of custom jvm arguments	

Examples

Generate JAX-WS source and classes in a separate JVM with separate directories, a custom wsdl location attribute, and a list of binding files from foo.wsdl:

```
<wsconsume
  fork="true"
  verbose="true"
  destdir="output"
  sourcedestdir="gen-src"
  keep="true"
  wsdllocation="handEdited.wsdl"
  wsdl="foo.wsdl">
  <binding dir="binding-files" includes="*.xml" excludes="bad.xml"/>
</wsconsume>
```

wsprovide

wsprovide is a command line tool, Maven plugin and Ant task that generates portable JAX-WS artifacts for a service endpoint implementation. It also has the option to "provide" the abstract contract for offline usage.

Command Line Tool

The command line tool has the following usage:

```
usage: wsprovide [options] <endpoint class name>
options:
  -h, --help                  Show this help message
  -k, --keep                  Keep/Generate Java source
  -w, --wsdl                  Enable WSDL file generation
  -a, --address The generated port soap:address in wsdl
  -c. --classpath=<path>      The classpath that contains the endpoint
  -o, --output=<directory>    The directory to put generated artifacts
  -r, --resource=<directory>  The directory to put resource artifacts
  -s, --source=<directory>    The directory to put Java source
  -e, --extension             Enable SOAP 1.2 binding extension
  -q, --quiet                 Be somewhat more quiet
  -t, --show-traces           Show full exception stack traces
```

Examples

Generating wrapper classes for portable artifacts in the "generated" directory:

```
wsprovide -o generated foo.Endpoint
```

Generating wrapper classes and WSDL in the "generated" directory

```
wsprovide -o generated -w foo.Endpoint
```

Using an endpoint that references other jars

```
wsprovide -o generated -c application1.jar:application2.jar foo.Endpoint
```

Maven Plugin

The *wsprovide* tools is included in the **org.jboss.ws.plugins:jaxws-tools-maven-plugin** plugin. The plugin has two goals for running the tool, *wsprovide* and *wsprovide-test*, which basically do the same during different Maven build phases (the former triggers the sources generation during *process-classes* phase, the latter during the *process-test-classes* one).

The *wsprovide* plugin has the following parameters:

Attribute	Description	Default
testClasspathElements	Each classpathElement provides a library file to be added to classpath	${project.compileClasspathElements} or ${project.testClasspathElements}
outputDirectory	The output directory for generated artifacts.	${project.build.outputDirectory} or ${project.build.testOutputDirectory}
resourceDirectory	The output directory for resource artifacts (WSDL/XSD).	${project.build.directory}/wsprovide/resources
sourceDirectory	The output directory for Java source.	${project.build.directory}/wsprovide/java
extension	Enable SOAP 1.2 binding extension.	false
generateWsdl	Whether or not to generate WSDL.	false
verbose	Enables more informational output about command progress.	false
portSoapAddress	The generated port soap:address in the WSDL	
endpointClass	**Service Endpoint Implementation.**	

Examples

You can use *wsprovide* in your own project build simply referencing the *maven-jaxws-tools-plugin* in the configured plugins in your *pom.xml* file.

The following example makes the plugin provide the wsdl file and artifact sources for the specified endpoint class:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.jboss.ws.plugins</groupId>
      <artifactId>jaxws-tools-maven-plugin</artifactId>
      <version>1.2.0.Beta1</version>
      <configuration>
        <verbose>true</verbose>
        <endpointClass>org.jboss.test.ws.plugins.tools.wsprovide.TestEndpoint</endpointClass>
        <generateWsdl>true</generateWsdl>
      </configuration>
      <executions>
        <execution>
          <goals>
            <goal>wsprovide</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

The following example does the same, but is meant for use in your own testsuite:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.jboss.ws.plugins</groupId>
      <artifactId>jaxws-tools-maven-plugin</artifactId>
      <version>1.2.0.Beta1</version>
      <configuration>
        <verbose>true</verbose>
        <endpointClass>org.jboss.test.ws.plugins.tools.wsprovide.TestEndpoint2</endpointClass>
        <generateWsdl>true</generateWsdl>
      </configuration>
      <executions>
        <execution>
          <goals>
            <goal>wsprovide-test</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

Plugin stack dependencyThe plugin itself does not have an explicit dependency to a JBossWS stack, as it's meant for being used with implementations of any supported version of the *JBossWS SPI*. So the user is expected to set a dependency in his own`pom.xml` to the desired *JBossWS* stack version. The plugin will rely on the that for using the proper tooling.

```
<dependencies>
  <dependency>
    <groupId>org.jboss.ws.cxf</groupId>
    <artifactId>jbossws-cxf-client</artifactId>
    <version>5.0.0.CR1</version>
  </dependency>
</dependencies>
```

Be careful when using this plugin with the Maven War Plugin as that include any project dependency into the generated application war archive. You might want to set `<scope>provided</scope>` for the *JBossWS* stack dependency to avoid that.

Up to version 1.1.2.Final, the *artifactId* of the plugin was **maven-jaxws-tools-plugin**.

Ant Task

The wsprovide ant task (*org.jboss.ws.tools.ant.WSProvideTask*) has the following attributes:

Attribute	Description	Default
fork	Whether or not to run the generation task in a separate VM.	true
keep	Keep/Enable Java source code generation.	false
destdir	The output directory for generated artifacts.	"output"
resourcedestdir	The output directory for resource artifacts (WSDL/XSD).	value of destdir
sourcedestdir	The output directory for Java source.	value of destdir
extension	Enable SOAP 1.2 binding extension.	false
genwsdl	Whether or not to generate WSDL.	false
address	The generated port soap:address in wsdl.	
verbose	Enables more informational output about command progress.	false
sei	**Service Endpoint Implementation.**	
classpath	The classpath that contains the service endpoint implementation.	"."

Examples

Executing wsprovide in verbose mode with separate output directories for source, resources, and classes:

```
<target name="test-wsproivde" depends="init">
  <taskdef name="wsprovide" classname="org.jboss.ws.tools.ant.WSProvideTask">
    <classpath refid="core.classpath"/>
  </taskdef>
  <wsprovide
    fork="false"
    keep="true"
    destdir="out"
    resourcedestdir="out-resource"
    sourcedestdir="out-source"
    genwsdl="true"
    verbose="true"
    sei="org.jboss.test.ws.jaxws.jsr181.soapbinding.DocWrappedServiceImpl">
    <classpath>
      <pathelement path="${tests.output.dir}/classes"/>
    </classpath>
  </wsprovide>
</target>
```

6.35.3 Advanced User Guide

Logging

Logging of inbound and outbound messages is a common need. Different approaches are available for achieving that:

WS Handler approach

A portable way of performing logging is writing a simple JAX-WS handler dumping the messages that are passed in it; the handler can be added to the desired client/endpoints (programmatically / using `@HandlerChain` JAX-WS annotation).

The predefined client and endpoint configuration mechanism allows user to add the logging handler to any client/endpoint or to some of them only (in which case the `@EndpointConfig` annotation / JBossWS API is required though).

Apache CXF approach

Apache CXF also comes with logging interceptors that can be easily used to log messages to the console or configured client/server log files. Those interceptors can be added to client, endpoint and buses in multiple ways:

System property

Setting the `org.apache.cxf.logging.enabled` system property to true causes the logging interceptors to be added to any `Bus` instance being created on the JVM.

On WildFly, the system property is easily set by adding what follows to the standalone / domain server configuration just after the extensions section:

```
<system-properties>
  <property name="org.apache.cxf.logging.enabled" value="true"/>
</system-properties>
```

Manual interceptor addition and logging feature

Logging interceptors can be selectively added to endpoints using the Apache CXF annotations `@org.apache.cxf.interceptor.InInterceptors` and `@org.apache.cxf.interceptor.OutInterceptors`. The same is achieved on client side by programmatically adding new instances of the logging interceptors to the client or the bus.

Alternatively, Apache CXF also comes with a `org.apache.cxf.feature.LoggingFeature` that can be used on clients and endpoints (either annotating them with `@org.apache.cxf.feature.Features` or directly with `@org.apache.cxf.annotations.Logging`).

Please refer to the Apache CXF documentation for more details.

* support

JBossWS includes most of the WS-* specification functionalities through the integration with Apache CXF. In particular, the whole WS-Security Policy framework is fully supported, enabling full contract driven configuration of complex features like WS-Security.

In details information available further down in this documentation book.

Address rewrite

JBossWS allows users to configure the *soap:address* attribute in the wsdl contract of deployed services.

Server configuration options

The configuration options are part of the webservices subsystem section of the application server domain model.

```
<subsystem xmlns="urn:jboss:domain:webservices:1.1"
xmlns:javaee="http://java.sun.com/xml/ns/javaee"
  xmlns:jaxwsconfig="urn:jboss:jbossws-jaxws-config:4.0">
  <wsdl-host>localhost</wsdl-host>
  <modify-wsdl-address>true</modify-wsdl-address>
<!--
  <wsdl-port>8080</wsdl-port>
  <wsdl-secure-port>8443</wsdl-secure-port>
-->
</subsystem>
```

If the content of *<soap:address>* in the wsdl is a valid URL, JBossWS will not rewrite it unless *modify-wsdl-address* is true. If the content of *<soap:address>* is not a valid URL instead, JBossWS will always rewrite it using the attribute values given below. Please note that the variable *${jboss.bind.address}* can be used to set the address which the application is bound to at each startup.

The wsdl-secure-port and wsdl-port attributes are used to explicitly define the ports to be used for rewriting the SOAP address. If these attributes are not set, the ports will be identified by querying the list of installed connectors. If multiple connectors are found the port of the first connector is used.

Dynamic rewrite

When the application server is bound to multiple addresses or non-trivial real-world network architectures cause request for different external addresses to hit the same endpoint, a static rewrite of the soap:address may not be enough. JBossWS allows for both the soap:address in the wsdl and the wsdl address in the console to be rewritten with the host use in the client request. This way, users always get the right wsdl address assuming they're connecting to an instance having the endpoint they're looking for. To trigger this behaviour, the *jbossws.undefined.host* value has to be specified for the *wsdl-host* element.

```
<wsdl-host>jbossws.undefined.host</wsdl-host>
<modify-wsdl-address>true</modify-wsdl-address>
```

Of course, when a confidential transport address is required, the addresses are always rewritten using https protocol and the port currently configured for the https/ssl connector.

Configuration through deployment descriptor

The `jboss-webservices.xml` deployment descriptor can be used to provide additional configuration for a given deployment. The expected location of it is:

- `META-INF/jboss-webservices.xml` for EJB webservice deployments
- `WEB-INF/jboss-webservices.xml` for POJO webservice deployments and EJB webservice endpoints bundled in `war` archives

The structure of file is the following (schemas are available here):

```
<webservices>
  <context-root/>?
  <config-name/>?
  <config-file/>?
  <property>*
    <name/>
    <value/>
  </property>
  <port-component>*
    <ejb-name/>
    <port-component-name/>
    <port-component-uri/>?
    <auth-method/>?
    <transport-guarantee/>?
    <secure-wsdl-access/>?
  </port-component>
  <webservice-description>*
    <webservice-description-name/>
    <wsdl-publish-location/>?
  </webservice-description>
</webservices>
```

context-root element

Element `<context-root>` can be used to customize context root of webservices deployment.

```
<webservices>
  <context-root>foo</context-root>
</webservices>
```

config-name and config-file elements

Elements `<config-name>` and `<config-file>` can be used to associate any endpoint provided in the deployment with a given endpoint configuration. Endpoint configuration are either specified in the referenced config file or in the WildFly domain model (webservices subsystem). For further details on the endpoint configurations and their management in the domain model, please see the related documentation.

```
<webservices>
  <config-name>Standard WSSecurity Endpoint</config-name>
  <config-file>META-INF/custom.xml</config-file>
</webservices>
```

property element

<property> elements can be used to setup simple property values to configure the ws stack behavior. Allowed property names and values are mentioned in the guide under related topics.

```
<property>
  <name>prop.name</name>
  <value>prop.value</value>
</property>
```

port-component element

Element <port-component> can be used to customize EJB endpoint target URI or to configure security related properties.

```
<webservices>
  <port-component>
    <ejb-name>TestService</ejb-name>
    <port-component-name>TestServicePort</port-component-name>
    <port-component-uri>/*</port-component-uri>
    <auth-method>BASIC</auth-method>
    <transport-guarantee>NONE</transport-guarantee>
    <secure-wsdl-access>true</secure-wsdl-access>
  </port-component>
</webservices>
```

webservice-description element

Element <webservice-description> can be used to customize (override) webservice WSDL publish location.

```
<webservices>
  <webservice-description>
    <webservice-description-name>TestService</webservice-description-name>
    <wsdl-publish-location>file:///bar/foo.wsdl</wsdl-publish-location>
  </webservice-description>
</webservices>
```

Schema validation of SOAP messages

Apache CXF has a feature for validating incoming and outgoing SOAP messages on both client and server side. The validation is performed against the relevant schema in the endpoint wsdl contract (server side) or the wsdl contract used for building up the service proxy (client side).

Schema validation can be turned on programmatically on client side

```
((BindingProvider)proxy).getRequestContext().put("schema-validation-enabled", true);
```

or using the `@org.apache.cxf.annotations.SchemaValidation` annotation on server side

```
import javax.jws.WebService;
import org.apache.cxf.annotations.SchemaValidation;

@WebService(...)
@SchemaValidation
public class ValidatingHelloImpl implements Hello {
   ...
}
```

Alternatively, any endpoint and client running in-container can be associated to a JBossWS predefined configuration having the `schema-validation-enabled` property set to `true` in the referenced config file.

Finally, JBossWS also allows for server-wide (default) setup of schema validation by using the *Standard-Endpoint-Config* and *Standard-Client-Config* special configurations (which apply to any client / endpoint unless a different configuration is specified for them)

```
<subsystem xmlns="urn:jboss:domain:webservices:1.2">
    ...
    <endpoint-config name="Standard-Endpoint-Config">
        <property name="schema-validation-enabled" value="true"/>
    </endpoint-config>
    ...
    <client-config name="Standard-Client-Config">
        <property name="schema-validation-enabled" value="true"/>
    </client-config>
</subsystem>
```

JAXB Introductions

As Kohsuke Kawaguchi wrote on his blog, one common complaint from the JAXB users is the lack of support for binding 3rd party classes. The scenario is this: you are trying to annotate your classes with JAXB annotations to make it XML bindable, but some of the classes are coming from libraries and JDK, and thus you cannot put necessary JAXB annotations on it.

To solve this JAXB has been designed to provide hooks for programmatic introduction of annotations to the runtime.

This is currently leveraged by the JBoss JAXB Introductions project, using which users can define annotations in XML and make JAXB see those as if those were in the class files (perhaps coming from 3rd party libraries).

Take a look at the JAXB Introductions page on the wiki and at the examples in the sources.

WSDL system properties expansion

See Published WSDL customization

Predefined client and endpoint configurations

- Overview
- Assigning configurations
    - Endpoint configuration assignment
    - Endpoint Configuration Deployment Descriptor
        - Application server configurations
            - Standard configurations
            - Handlers classloading
            - Examples
        - EndpointConfig annotation
        - JAXWS Feature
        - Explicit setup through API
    - Automatic configuration from default descriptors
    - Automatic configuration assignment from container setup

Overview

JBossWS permits extra setup configuration data to be predefined and associated with an endpoint or a client. Configurations can include JAX-WS handlers and key/value property declarations that control JBossWS and Apache CXF internals. Predefined configurations can be used for JAX-WS client and JAX-WS endpoint setup.

Configurations can be defined in the webservice subsystem and in an application's deployment descriptor file. There can be many configuration definitions in the webservice subsystem and in an application. Each configuration must have a name that is unique within the server. Configurations defined in an application are local to the application. Endpoint implementations declare the use of a specific configuration through the use of the `org.jboss.ws.api.annotation.EndpointConfig` annotation. An endpoint configuration defined in the webservices subsystem is available to all deployed applications on the server container and can be referenced by name in the annotation. An endpoint configuration defined in an application must be referenced by both deployment descriptor file name and configuration name by the annotation.

Handlers

Each endpoint configuration may be associated with zero or more PRE and POST handler chains. Each handler chain may include JAXWS handlers. For outbound messages the PRE handler chains are executed before any handler that is attached to the endpoint using the standard means, such as with annotation @HandlerChain, and POST handler chains are executed after those objects have executed. For inbound messages the POST handler chains are executed before any handler that is attached to the endpoint using the standard means and the PRE handler chains are executed after those objects have executed.

```
* Server inbound messages
Client --> ... --> POST HANDLER --> ENDPOINT HANDLERS --> PRE HANDLERS --> Endpoint

* Server outbound messages
Endpoint --> PRE HANDLER --> ENDPOINT HANDLERS --> POST HANDLERS --> ... --> Client
```

The same applies for client configurations.

Properties

Key/value properties are used for controlling both some Apache CXF internals and some JBossWS options. Specific supported values are mentioned where relevant in the rest of the documentation.

Assigning configurations

Endpoints and clients are assigned configuration through different means. Users can explicitly require a given configuration or rely on container defaults. The assignment process can be split up as follows:

- Explicit assignment through annotations (for endpoints) or API programmatic usage (for clients)
- Automatic assignment of configurations from default descriptors
- Automatic assignment of configurations from container

Endpoint configuration assignment

The explicit configuration assignment is meant for developer that know in advance their endpoint or client has to be setup according to a specified configuration. The configuration is either coming from a descriptor that is included in the application deployment, or is included in the application server webservices subsystem management model.

Endpoint Configuration Deployment Descriptor

Java EE archives that can contain JAX-WS client and endpoint implementations can also contain predefined client and endpoint configuration declarations. All endpoint/client configuration definitions for a given archive must be provided in a single deployment descriptor file, which must be an implementation of schema jbossws-jaxws-config. Many endpoint/client configurations can be defined in the deployment descriptor file. Each configuration must have a name that is unique within the server on which the application is deployed. The configuration name can't be referred to by endpoint/client implementations outside the application. Here is an example of a descriptor, containing two endpoint configurations:

```
<?xml version="1.0" encoding="UTF-8"?>
<jaxws-config xmlns="urn:jboss:jbossws-jaxws-config:4.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:javaee="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="urn:jboss:jbossws-jaxws-config:4.0 schema/jbossws-jaxws-config_4_0.xsd">
<endpoint-config>
<config-name>org.jboss.test.ws.jaxws.jbws3282.Endpoint4Impl</config-name>
<pre-handler-chains>
<javaee:handler-chain>
<javaee:handler>
<javaee:handler-name>Log Handler</javaee:handler-name>
<javaee:handler-class>org.jboss.test.ws.jaxws.jbws3282.LogHandler</javaee:handler-class>
</javaee:handler>
</javaee:handler-chain>
</pre-handler-chains>
<post-handler-chains>
<javaee:handler-chain>
<javaee:handler>
<javaee:handler-name>Routing Handler</javaee:handler-name>
<javaee:handler-class>org.jboss.test.ws.jaxws.jbws3282.RoutingHandler</javaee:handler-class>
</javaee:handler>
</javaee:handler-chain>
</post-handler-chains>
</endpoint-config>
<endpoint-config>
<config-name>EP6-config</config-name>
<post-handler-chains>
<javaee:handler-chain>
<javaee:handler>
<javaee:handler-name>Authorization Handler</javaee:handler-name>
<javaee:handler-class>org.jboss.test.ws.jaxws.jbws3282.AuthorizationHandler</javaee:handler-class>
```

Similarly, client configurations can be specified in descriptors (still implementing the schema mentioned above):

```
<?xml version="1.0" encoding="UTF-8"?>
<jaxws-config xmlns="urn:jboss:jbossws-jaxws-config:4.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:javaee="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="urn:jboss:jbossws-jaxws-config:4.0 schema/jbossws-jaxws-config_4_0.xsd">
<client-config>
<config-name>Custom Client Config</config-name>
<pre-handler-chains>
<javaee:handler-chain>
<javaee:handler>
<javaee:handler-name>Routing Handler</javaee:handler-name>
<javaee:handler-class>org.jboss.test.ws.jaxws.clientConfig.RoutingHandler</javaee:handler-class>
</javaee:handler>
<javaee:handler>
<javaee:handler-name>Custom Handler</javaee:handler-name>
<javaee:handler-class>org.jboss.test.ws.jaxws.clientConfig.CustomHandler</javaee:handler-class>
</javaee:handler>
</javaee:handler-chain>
</pre-handler-chains>
</client-config>
<client-config>
<config-name>Another Client Config</config-name>
<post-handler-chains>
<javaee:handler-chain>
<javaee:handler>
<javaee:handler-name>Routing Handler</javaee:handler-name>
<javaee:handler-class>org.jboss.test.ws.jaxws.clientConfig.RoutingHandler</javaee:handler-class>
</javaee:handler>
</javaee:handler-chain>
</post-handler-chains>
</client-config>
</jaxws-config>
```

Application server configurations

WildFly allows declaring JBossWS client and server predefined configurations in the *webservices* subsystem section of the server model. As a consequence it is possible to declare server-wide handlers to be added to the chain of each endpoint or client assigned to a given configuration.

Please refer to the WildFly documentation for details on managing the *webservices* subsystem such as adding, removing and modifying handlers and properties.

The allowed contents in the *webservices* subsystem are defined by the schema included in the application server.

Standard configurations

Clients running in-container as well as endpoints are assigned standard configurations by default. The defaults are used unless different configurations are set as described on this page. This enables administrators to tune the default handler chains for client and endpoint configurations. The names of the default client and endpoint configurations, used in the webservices subsystem are `Standard-Client-Config` and `Standard-Endpoint-Config` respectively.

Handlers classloading

When setting a server-wide handler, please note the handler class needs to be available through each ws deployment classloader. As a consequence proper module dependencies might need to be specified in the deployments that are going to leverage a given predefined configuration. A shortcut is to add a dependency to the module containing the handler class in one of the modules which are already automatically set as dependencies to any deployment, for instance `org.jboss.ws.spi`.

Examples

JBoss AS 7.2 default configurations

```
<subsystem xmlns="urn:jboss:domain:webservices:2.0">
<!-- ... -->
<endpoint-config name="Standard-Endpoint-Config"/>
<endpoint-config name="Recording-Endpoint-Config">
<pre-handler-chain name="recording-handlers" protocol-bindings="##SOAP11_HTTP ##SOAP11_HTTP_MTOM
##SOAP12_HTTP ##SOAP12_HTTP_MTOM">
<handler name="RecordingHandler" class="org.jboss.ws.common.invocation.RecordingServerHandler"/>
</pre-handler-chain>
</endpoint-config>
<client-config name="Standard-Client-Config"/>
</subsystem>
```

A configuration file for a deployment specific ws-security endpoint setup

```
<jaxws-config xmlns="urn:jboss:jbossws-jaxws-config:4.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:javaee="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="urn:jboss:jbossws-jaxws-config:4.0 schema/jbossws-jaxws-config_4_0.xsd">
<endpoint-config>
<config-name>Custom WS-Security Endpoint</config-name>
<property>
<property-name>ws-security.signature.properties</property-name>
<property-value>bob.properties</property-value>
</property>
<property>
<property-name>ws-security.encryption.properties</property-name>
<property-value>bob.properties</property-value>
</property>
<property>
<property-name>ws-security.signature.username</property-name>
<property-value>bob</property-value>
</property>
<property>
<property-name>ws-security.encryption.username</property-name>
<property-value>alice</property-value>
</property>
<property>
<property-name>ws-security.callback-handler</property-name>
<property-value>org.jboss.test.ws.jaxws.samples.wsse.policy.basic.KeystorePasswordCallback</proper
```

JBoss AS 7.2 default configurations modified to default to SOAP messages schema-validation on

```
<subsystem xmlns="urn:jboss:domain:webservices:2.0">
<!-- ... -->
<endpoint-config name="Standard-Endpoint-Config">
<property name="schema-validation-enabled" value="true"/>
</endpoint-config>
<!-- ... -->
<client-config name="Standard-Client-Config">
<property name="schema-validation-enabled" value="true"/>
</client-config>
</subsystem>
```

EndpointConfig annotation

Once a configuration is available to a given application, the `org.jboss.ws.api.annotation.EndpointConfig` annotation is used to assign an endpoint configuration to a JAX-WS endpoint implementation. When assigning a configuration that is defined in the webservices subsystem only the configuration name is specified. When assigning a configuration that is defined in the application, the relative path to the deployment descriptor and the configuration name must be specified.

```
@EndpointConfig(configFile = "WEB-INF/my-endpoint-config.xml", configName = "Custom WS-Security
Endpoint")
public class ServiceImpl implements ServiceIface
{
public String sayHello()
{
return "Secure Hello World!";
}
}
```

JAXWS Feature

The most practical way of setting a configuration is using `org.jboss.ws.api.configuration.ClientConfigFeature`, a JAXWS `Feature` extension provided by JBossWS:

```
import org.jboss.ws.api.configuration.ClientConfigFeature;

...

Service service = Service.create(wsdlURL, serviceName);
Endpoint port = service.getPort(Endpoint.class, new
ClientConfigFeature("META-INF/my-client-config.xml", "Custom Client Config"));
port.echo("Kermit");

... or ....

port = service.getPort(Endpoint.class, new ClientConfigFeature("META-INF/my-client-config.xml",
"Custom Client Config"), true); //setup properties too from the configuration
port.echo("Kermit");
... or ...

port = service.getPort(Endpoint.class, new ClientConfigFeature(null, testConfigName)); //reads
from current container configurations if available
port.echo("Kermit");
```

JBossWS parses the specified configuration file. The configuration file must be found as a resource by the classloader of the current thread. The jbossws-jaxws-config schema defines the descriptor contents and is included in the *jbossws-spi* artifact.

Explicit setup through API

Alternatively, JBossWS API comes with facility classes that can be used for assigning configurations when building a client. JAXWS handlers read from client configurations as follows:

```
import org.jboss.ws.api.configuration.ClientConfigUtil;
import org.jboss.ws.api.configuration.ClientConfigurer;

...

Service service = Service.create(wsdlURL, serviceName);
Endpoint port = service.getPort(Endpoint.class);
BindingProvider bp = (BindingProvider)port;
ClientConfigUtil.setConfigHandlers(bp, "META-INF/my-client-config.xml", "Custom Client Config
1");
port.echo("Kermit");

...

ClientConfigurer configurer = ClientConfigUtil.resolveClientConfigurer();
configurer.setConfigHandlers(bp, "META-INF/my-client-config.xml", "Custom Client Config 2");
port.echo("Kermit");

...

configurer.setConfigHandlers(bp, "META-INF/my-client-config.xml", "Custom Client Config 3");
port.echo("Kermit");


...

configurer.setConfigHandlers(bp, null, "Container Custom Client Config"); //reads from current
container configurations if available
port.echo("Kermit");
```

... similarly, properties are read from client configurations as follows:

```
import org.jboss.ws.api.configuration.ClientConfigUtil;
import org.jboss.ws.api.configuration.ClientConfigurer;

...

Service service = Service.create(wsdlURL, serviceName);
Endpoint port = service.getPort(Endpoint.class);

ClientConfigUtil.setConfigProperties(port, "META-INF/my-client-config.xml", "Custom Client
Config 1");
port.echo("Kermit");

...

ClientConfigurer configurer = ClientConfigUtil.resolveClientConfigurer();
configurer.setConfigProperties(port, "META-INF/my-client-config.xml", "Custom Client Config 2");
port.echo("Kermit");

...

configurer.setConfigProperties(port, "META-INF/my-client-config.xml", "Custom Client Config 3");
port.echo("Kermit");


...

configurer.setConfigProperties(port, null, "Container Custom Client Config"); //reads from
current container configurations if available
port.echo("Kermit");
```

The default `ClientConfigurer` implementation parses the specified configuration file, if any, after having resolved it as a resources using the current thread context classloader. The jbossws-jaxws-config schema defines the descriptor contents and is included in the *jbossws-spi* artifact.

Automatic configuration from default descriptors

In some cases, the application developer might not be aware of the configuration that will need to be used for its client and endpoint implementation, perhaps because that's a concern of the application deployer. In other cases, explicit usage (compile time dependency) of JBossWS API might not be accepted. To cope with such scenarios, JBossWS allows including default client (`jaxws-client-config.xml`) and endpoint (`jaxws-endpoint-config.xml`) descriptor within the application (in its root), which are parsed for getting configurations any time a configuration file name is not specified.

If the configuration name is also not specified, JBossWS automatically looks for a configuration named the same as

- the endpoint implementation class (full qualified name), in case of JAX-WS endpoints;
- the service endpoint interface (full qualified name), in case of JAX-WS clients.

No automatic configuration name is selected for `Dispatch` clients.

So, for instance, an endpoint implementation class `org.foo.bar.EndpointImpl` for which no pre-defined configuration is explicitly set will cause JBossWS to look for a *org.foo.bar.EndpointImpl* named configuration within a *jaxws-endpoint-config.xml* descriptor in the root of the application deployment. Similarly, on client side, a client proxy implementing `org.foo.bar.Endpoint` interface (SEI) will have the setup read from a *org.foo.bar.Endpoint* named configuration in *jaxws-client-config.xml* descriptor.

Automatic configuration assignment from container setup

JBossWS fall-backs to getting predefined configurations from the container setup whenever no explicit configuration has been provided and the default descriptors are either not available or do not contain relevant configurations. This gives additional control on the JAX-WS client and endpoint setup to administrators, as the container setup can be managed independently from the deployed applications. JBossWS hence accesses the webservices subsystem the same as explained above for explicitly named configuration; the default configuration names used for look are

- the endpoint implementation class (full qualified name), in case of JAX-WS endpoints;
- the service endpoint interface (full qualified name), in case of JAX-WS clients.
  `Dispatch` clients are not automatically configured. If no configuration is found using names computed as above, the `Standard-Client-Config` and `Standard-Endpoint-Config` configurations are used for clients and endpoints respectively

Authentication

Authentication

Here the simplest way to authenticate a web service user with JBossWS is explained.

First we secure the access to the SLSB as we would do for normal (non web service) invocations: this can be easily done through the @RolesAllowed, @PermitAll, @DenyAll annotation. The allowed user roles can be set with these annotations both on the bean class and on any of its business methods.

```
@Stateless
@RolesAllowed("friend")
public class EndpointEJB implements EndpointInterface
{
  ...
}
```

Similarly POJO endpoints are secured the same way as we do for normal web applications in web.xml:

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>All resources</web-resource-name>
    <url-pattern>/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>friend</role-name>
  </auth-constraint>
</security-constraint>

<security-role>
  <role-name>friend</role-name>
</security-role>
```

Specify the security domain

Next, specify the security domain for this deployment. This is performed using the `@SecurityDomain` annotation for EJB3 endpoints

```
@Stateless
@SecurityDomain("JBossWS")
@RolesAllowed("friend")
public class EndpointEJB implements EndpointInterface
{
  ...
}
```

or modifying the jboss-web.xml for POJO endpoints

```
<jboss-web>
<security-domain>JBossWS</security-domain>
</jboss-web>
```

The security domain as well as its the authentication and authorization mechanisms are defined differently depending on the application server version in use.

Use BindingProvider to set principal/credential

A web service client may use the `javax.xml.ws.BindingProvider` interface to set the username/password combination

```
URL wsdlURL = new
File("resources/jaxws/samples/context/WEB-INF/wsdl/TestEndpoint.wsdl").toURL();
QName qname = new QName("http://org.jboss.ws/jaxws/context", "TestEndpointService");
Service service = Service.create(wsdlURL, qname);
port = (TestEndpoint)service.getPort(TestEndpoint.class);

BindingProvider bp = (BindingProvider)port;
bp.getRequestContext().put(BindingProvider.USERNAME_PROPERTY, "kermit");
bp.getRequestContext().put(BindingProvider.PASSWORD_PROPERTY, "thefrog");
```

Using HTTP Basic Auth for security

To enable HTTP Basic authentication you use the `@WebContext` annotation on the bean class

```
@Stateless
@SecurityDomain("JBossWS")
@RolesAllowed("friend")
@WebContext(contextRoot="/my-cxt", urlPattern="/*", authMethod="BASIC",
transportGuarantee="NONE", secureWSDLAccess=false)
public class EndpointEJB implements EndpointInterface
{
  ...
}
```

For POJO endpoints, we modify the *web.xml* adding the auth-method element:

```
<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>Test Realm</realm-name>
</login-config>
```

JASPI Authentication

A Java Authentication SPI (JASPI) provider can be configured in WildFly security subsystem to authenticate SOAP messages:

```
<security-domain name="jaspi">
<authentication-jaspi>
<login-module-stack name="jaas-lm-stack">
<login-module code="UsersRoles" flag="required">
<module-option name="usersProperties" value="jbossws-users.properties"/>
<module-option name="rolesProperties" value="jbossws-roles.properties"/>
</login-module>
</login-module-stack>
<auth-module code="org.jboss.wsf.stack.cxf.jaspi.module.UsernameTokenServerAuthModule"
login-module-stack-ref="jaas-lm-stack"/>
</authentication-jaspi>
</security-domain>
```

For further information on configuring security domains in WildFly, please refer to here.

Here `org.jboss.wsf.stack.cxf.jaspi.module.UsernameTokenServerAuthModule` is the class implementing `javax.security.auth.message.module.ServerAuthModule`, which delegates to the proper login module to perform authentication using the credentials from WS-Security UsernameToken in the incoming SOAP message. Alternative implementations of `ServerAuthModule` can be implemented and configured.

To enable JASPI authentication, the endpoint deployment needs to specify the security domain to use; that can be done in two different ways:

- Setting the `jaspi.security.domain` property in the `jboss-webservices.xml` descriptor

```
<?xml version="1.1" encoding="UTF-8"?>
<webservices
xmlns="http://www.jboss.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
version="1.2"
xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee">

<property>
<name>jaspi.security.domain</name>
<value>jaspi</value>
</property>

</webservices>
```

- Referencing (through `@EndpointConfig` annotation) an endpoint config that sets the `jaspi.security.domain` property

```
@EndpointConfig(configFile = "WEB-INF/jaxws-endpoint-config.xml", configName =
"jaspiSecurityDomain")
public class ServiceEndpointImpl implements ServiceIface {
```

The `jaspi.security.domain` property is specified as follows in the referenced descriptor:

```
<?xml version="1.0" encoding="UTF-8"?>
<jaxws-config xmlns="urn:jboss:jbossws-jaxws-config:4.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:javaee="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="urn:jboss:jbossws-jaxws-config:4.0 schema/jbossws-jaxws-config_4_0.xsd">
<endpoint-config>
<config-name>jaspiSecurityDomain</config-name>
<property>
<property-name>jaspi.security.domain</property-name>
<property-value>jaspi</property-value>
</property>
</endpoint-config>
</jaxws-config>
```

If the JASPI security domain is specified in both `jboss-webservices.xml` and config file referenced by `@EndpointConfig` annotation, the JASPI security domain specified in `jboss-webservices.xml` will take precedence.

Apache CXF integration

JBossWS integration layer with Apache CXF

All JAX-WS functionalities provided by JBossWS on top of WildFly are currently served through a proper integration of the JBoss Web Services stack with most of the Apache CXF project modules.

Apache CXF is an open source services framework. It allows building and developing services using frontend programming APIs (including JAX-WS), with services speaking a variety of protocols such as SOAP and XML/HTTP over a variety of transports such as HTTP and JMS.

The integration layer (*JBossWS-CXF* in short hereafter) is mainly meant for:

- allowing using standard webservices APIs (including JAX-WS) on WildFly; this is performed internally leveraging Apache CXF without requiring the user to deal with it;
- allowing using Apache CXF advanced features (including WS-*) on top of WildFly without requiring the user to deal with / setup / care about the required integration steps for running in such a container.

In order for achieving the goals above, the JBossWS-CXF integration supports the JBoss ws endpoint deployment mechanism and comes with many internal customizations on top of Apache CXF.

In the next sections a list of technical suggestions and notes on the integration is provided; please also refer to the Apache CXF official documentation for in-depth details on the CXF architecture.

Building WS applications the JBoss way

The Apache CXF client and endpoint configuration as explained in the Apache CXF official user guide is heavily based on Spring. Apache CXF basically parses Spring `cxf.xml` descriptors; those may contain any basic bean plus specific ws client and endpoint beans which CXF has custom parsers for. Apache CXF can be used to deploy webservice endpoints on any servlet container by including its libraries in the deployment; in such a scenario Spring basically serves as a convenient configuration option, given direct Apache CXF API usage won't be very handy. Similar reasoning applies on client side, where a Spring based descriptor offers a shortcut for setting up Apache CXF internals.

This said, nowadays almost any Apache CXF functionality can be configured and used through direct API usage, without Spring. As a consequence of that and given the considerations in the sections below, the JBossWS integration with Apache CXF does not rely on Spring descriptors.

Portable applications

WildFly is much more then a servlet container; it actually provides users with a fully compliant target platform for Java EE applications.

Generally speaking, *users are encouraged to write portable applications* by relying only on *JAX-WS specification* whenever possible. That would by the way ensure easy migrations to and from other compliant platforms. Being a Java EE container, WildFly already comes with a JAX-WS compliant implementation, which is basically Apache CXF plus the JBossWS-CXF integration layer. So users just need to write their JAX-WS application; *no need for embedding any Apache CXF or any ws related dependency library in user deployments*. Please refer to the JAX-WS User Guide section of the documentation for getting started.

WS-* usage (including WS-Security, WS-Addressing, WS-ReliableMessaging, ...) should also be configured in the most portable way; that is by *relying on proper WS-Policy assertions* on the endpoint WSDL contracts, so that client and endpoint configuration is basically a matter of setting few ws context properties. The WS-* related sections of this documentation cover all the details on configuring applications making use of WS-* through policies.

As a consequence of the reasoning above, the JBossWS-CXF integration is currently built directly on the Apache CXF API and aims at allowing users to configure webservice clients and endpoints *without Spring descriptors*.

Direct Apache CXF API usage

Whenever users can't really meet their application requirements with JAX-WS plus WS-Policy, it is of course still possible to rely on direct Apache CXF API usage (given that's included in the AS), loosing the Java EE portability of the application. That could be the case of a user needing specific Apache CXF functionalities, or having to consume WS-* enabled endpoints advertised through legacy wsdl contracts without WS-Policy assertions.

On server side, direct Apache CXF API usage might not be always possible or end up being not very easy. For this reason, the JBossWS integration comes with a convenient alternative through customization options in the `jboss-webservices.xml` descriptor described below on this page. Properties can be declared in `jboss-webservices.xml` to control Apache CXF internals like *interceptors*, *features*, etc.

Bus usage

Creating a Bus instance

Most of the Apache CXF features are configurable using the `org.apache.cxf.Bus` class. While for basic JAX-WS usage the user might never need to explicitly deal with Bus, using Apache CXF specific features generally requires getting a handle to a `org.apache.cxf.Bus` instance. This can happen on client side as well as in a ws endpoint or handler business code.

New Bus instances are produced by the currently configured `org.apache.cxf.BusFactory` implementation the following way:

```
Bus bus = BusFactory.newInstance().createBus();
```

The algorithm for selecting the actual implementation of `BusFactory` to be used leverages the Service API, basically looking for optional configurations in *META-INF/services/...* location using the current thread context classloader. JBossWS-CXF integration comes with its own implementation of `BusFactory`, `org.jboss.wsf.stack.cxf.client.configuration.JBossWSBusFactory`, that allows for seamless setup of JBossWS customizations on top of Apache CXF. So, assuming the JBossWS-CXF libraries are available in the current thread context classloader, the `JBossWSBusFactory` is *automatically* retrieved by the `BusFactory.newInstance()` call above.

JBossWS users willing to explicitly use functionalities of `org.apache.cxf.bus.CXFBusFactory`, get the same API with JBossWS additions through `JBossWSBusFactory`:

```
Map<Class, Object> myExtensions = new HashMap<Class, Object>();
myExtensions.put(...);
Bus bus = new JBossWSBusFactory().createBus(myExtensions);
```

Using existing Bus instances

Apache CXF keeps reference to a global default `Bus` instance as well as to a thread default bus for each thread. That is performed through static members in `org.apache.cxf.BusFactory`, which also comes with the following methods in the public API:

```
public static synchronized Bus getDefaultBus()
public static synchronized Bus getDefaultBus(boolean createIfNeeded)
public static synchronized void setDefaultBus(Bus bus)
public static Bus getThreadDefaultBus()
public static Bus getThreadDefaultBus(boolean createIfNeeded)
public static void setThreadDefaultBus(Bus bus)
```

Please note that the default behaviour of `getDefaultBus()` / `getDefaultBus(true)` / `getThreadDefaultBus()` / `getThreadDefaultBus(true)` is to create a new Bus instance if that's not set yet. Moreover *getThreadDefaultBus()* and *getThreadDefaultBus(true)* first fallback to retrieving the configured global default bus before actually trying creating a new instance (and the created new instance is set as global default bus if that was not set there yet).

The drawback of this mechanism (which is basically fine in JSE environment) is that when running in WildFly container you need to be careful in order not to (mis)use a bus over multiple applications (assuming the Apache CXF classes are loaded by the same classloader, which is currently the case with WildFly).

Here is a list of general suggestions to avoid problems when running in-container:

- forget about the global default bus; you don't need that, so don't do `getDefaultBus()` / `getDefaultBus(true)` / `setDefaultBus()` in your code;
- avoid `getThreadDefaultBus()` / `getThreadDefaultBus(true)` unless you already know for sure the default bus is already set;
- keep in mind thread pooling whenever you customize a thread default bus instance (for instance adding bus scope interceptors, ...), as that thread and bus might be later reused; so either shutdown the bus when you're done or explicitly remove it from the BusFactory thread association.

Finally, remember that each time you explictly create a new Bus instance (factory.createBus()) that is set as thread default bus and global default bus if those are not set yet. The JAXWS `Provider` implementation also creates `Bus` instances internally, in particular the JBossWS version of JAXWS `Provider` makes sure the default bus is never internally used and instead a new `Bus` is created if required (more details on this in the next paragraph).

Bus selection strategies for JAXWS clients

JAXWS clients require an Apache CXF Bus to be available; the client is registered within the Bus and the Bus affects the client behavior (e.g. through the configured CXF interceptors). The way a bus is internally selected for serving a given JAXWS client is very important, especially for in-container clients; for this reason, JBossWS users can choose the preferred Bus selection strategy. The strategy is enforced in the `javax.xml.ws.spi.Provider` implementation from the JBossWS integration, being that called whenever a JAXWS `Service` (client) is requested.

Thread bus strategy (THREAD_BUS)
Each time the vanilla JAXWS api is used to create a Bus, the JBossWS-CXF integration will automatically make sure a Bus is currently associated to the current thread in the BusFactory. If that's not the case, a new Bus is created and linked to the current thread (to prevent the user from relying on the default Bus). The Apache CXF engine will then create the client using the current thread Bus.

This is the default strategy, and the most straightforward one in Java SE environments; it lets users automatically reuse a previously created Bus instance and allows using customized Bus that can possibly be created and associated to the thread before building up a JAXWS client.

The drawback of the strategy is that the link between the Bus instance and the thread needs to be eventually cleaned up (when not needed anymore). This is really evident in a Java EE environment (hence when running in-container), as threads from pools (e.g. serving web requests) are re-used.

When relying on this strategy, the safest approach to be sure of cleaning up the link is to surround the JAXWS client with a `try/finally` block as below:

```
try {
Service service = Service.create(wsdlURL, serviceQName);
MyEndpoint port = service.getPort(MyEndpoint.class);
//...
} finally {
BusFactory.setThreadDefaultBus(null);
// OR (if you don't need the bus and the client anymore)
 Bus bus = BusFactory.getThreadDefaultBus(false);
bus.shutdown(true);
}
```

New bus strategy (NEW_BUS)
Another strategy is to have the JAXWS Provider from the JBossWS integration create a new Bus each time a JAXWS client is built. The main benefit of this approach is that a fresh bus won't rely on any formerly cached information (e.g. cached WSDL / schemas) which might have changed after the previous client creation. The main drawback is of course worse performance as the Bus creation takes time.

If there's a bus already associated to the current thread before the JAXWS client creation, that is automatically restored when returning control to the user; in other words, the newly created bus will be used only for the created JAXWS client but won't stay associated to the current thread at the end of the process. Similarly, if the thread was not associated to any bus before the client creation, no bus will be associated to the thread at the end of the client creation.

Thread context classloader bus strategy (TCCL_BUS)
The last strategy is to have the bus created for serving the client be associated to the current thread context classloader (TCCL). That basically means the same Bus instance is shared by JAXWS clients running when the same TCCL is set. This is particularly interesting as each web application deployment usually has its own context classloader, so this strategy is possibly a way to keep the number of created Bus instances bound to the application number in WildFly container.

If there's a bus already associated to the current thread before the JAXWS client creation, that is automatically restored when returning control to the user; in other words, the bus corresponding to the current thread context classloader will be used only for the created JAXWS client but won't stay associated to the current thread at the end of the process. If the thread was not associated to any bus before the client creation, a new bus will be created (and later user for any other client built with this strategy and the same TCCL in place); no bus will be associated to the thread at the end of the client creation.
Strategy configuration
Users can request a given Bus selection strategy to be used for the client being built by specifying one of the following JBossWS features (which extend `javax.xml.ws.WebServiceFeature`):

Feature	Strategy
`org.jboss.wsf.stack.cxf.client.UseThreadBusFeature`	THREAD_BUS
`org.jboss.wsf.stack.cxf.client.UseNewBusFeature`	NEW_BUS
`org.jboss.wsf.stack.cxf.client.UseTCCLBusFeature`	TCCL_BUS

The feature is specified as follows:

```
Service service = Service.create(wsdlURL, serviceQName, new UseThreadBusFeature());
```

If no feature is explicitly specified, the system default strategy is used, which can be modified through the `org.jboss.ws.cxf.jaxws-client.bus.strategy` system property when starting the JVM. The valid values for the property are `THREAD_BUS`, `NEW_BUS` and `TCCL_BUS`. The default is `THREAD_BUS`.

Server Side Integration Customization

The JBossWS-CXF server side integration takes care of internally creating proper Apache CXF structures (including a `Bus` instance, of course) for the provided ws deployment. Should the deployment include multiple endpoints, those would all live within the same Apache CXF Bus, which would of course be completely separated by the other deployments' bus instances.

While JBossWS sets sensible defaults for most of the Apache CXF configuration options on server side, users might want to fine tune the `Bus` instance that's created for their deployment; a `jboss-webservices.xml` descriptor can be used for deployment level customizations.

Deployment descriptor properties

The `jboss-webservices.xml` descriptor can be used to provide property values.

```
<webservices xmlns="http://www.jboss.com/xml/ns/javaee" version="1.2">
  ...
  <property>
    <name>...</name>
    <value>...</value>
  </property>
  ...
</webservices>
```

JBossWS-CXF integration comes with a set of allowed property names to control Apache CXF internals.

WorkQueue configuration

Apache CXF uses WorkQueue instances for dealing with some operations (e.g. @Oneway requests processing). A WorkQueueManager is installed in the Bus as an extension and allows for adding / removing queues as well as controlling the existing ones.

On server side, queues can be provided by using the `cxf.queue.<queue-name>.*` properties in `jboss-webservices.xml` (e.g. `cxf.queue.default.maxQueueSize` for controlling the max queue size of the `default` workqueue). At deployment time, the JBossWS integration can add new instances of AutomaticWorkQueueImpl to the currently configured WorkQueueManager; the properties below are used to fill in parameter into the AutomaticWorkQueueImpl constructor:

Property	Default value
`cxf.queue.<queue-name>.maxQueueSize`	256
`cxf.queue.<queue-name>.initialThreads`	0
`cxf.queue.<queue-name>.highWaterMark`	25
`cxf.queue.<queue-name>.lowWaterMark`	5
`cxf.queue.<queue-name>.dequeueTimeout`	120000

Policy alternative selector

The Apache CXF policy engine supports different strategies to deal with policy alternatives. JBossWS-CXF integration currently defaults to the MaximalAlternativeSelector, but still allows for setting different selector implementation using the `cxf.policy.alternativeSelector` property in `jboss-webservices.xml`.

MBean management

Apache CXF allows managing its MBean objects that are installed into the WildFly MBean server. The feature is enabled on a deployment basis through the `cxf.management.enabled` property in `jboss-webservices.xml`. The `cxf.management.installResponseTimeInterceptors` property can also be used to control installation of CXF response time interceptors, which are added by default when enabling MBean management, but might not be desired in some cases. Here is an example:

```
<webservices xmlns="http://www.jboss.com/xml/ns/javaee" version="1.2">
<property>
<name>cxf.management.enabled</name>
<value>true</value>
</property>
<property>
<name>cxf.management.installResponseTimeInterceptors</name>
<value>false</value>
</property>
</webservices>
```

Schema validation

Schema validation of exchanged messages can also be enabled in `jboss-webservices.xml`. Further details available here.

Interceptors

The `jboss-webservices.xml` descriptor also allows specifying the `cxf.interceptors.in` and `cxf.interceptors.out` properties; those allows declaring interceptors to be attached to the Bus instance that's created for serving the deployment.

```
<?xml version="1.1" encoding="UTF-8"?>
<webservices
xmlns="http://www.jboss.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
version="1.2"
xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee">

<property>
<name>cxf.interceptors.in</name>
<value>org.jboss.test.ws.jaxws.cxf.interceptors.BusInterceptor</value>
</property>
<property>
<name>cxf.interceptors.out</name>
<value>org.jboss.test.ws.jaxws.cxf.interceptors.BusCounterInterceptor</value>
</property>
</webservices>
```

Features

The `jboss-webservices.xml` descriptor also allows specifying the `cxf.features` property; that allows declaring features to be attached to any endpoint belonging to the Bus instance that's created for serving the deployment.

```
<?xml version="1.1" encoding="UTF-8"?>
<webservices
xmlns="http://www.jboss.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
version="1.2"
xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee">

<property>
<name>cxf.features</name>
<value>org.apache.cxf.feature.FastInfosetFeature</value>
</property>
</webservices>
```

Discovery enablement

WS-Discovery support can be turned on in `jboss-webservices` for the current deployment. Further details available here.

Apache CXF interceptors

Apache CXF supports declaring interceptors using one of the following approaches:

- Annotation usage on endpoint classes (`@org.apache.cxf.interceptor.InInterceptor`, `@org.apache.cxf.interceptor.OutInterceptor`)
- Direct API usage on client side (through the `org.apache.cxf.interceptor.InterceptorProvider` interface)
- Spring descriptor usage (*cxf.xml*)

As the Spring descriptor usage is not supported, the JBossWS integration adds an additional descriptor based approach to avoid requiring modifications to the actual client/endpoint code. Users can declare interceptors within predefined client and endpoint configurations by specifying a list of interceptor class names for the `cxf.interceptors.in` and `cxf.interceptors.out` properties.

```
<?xml version="1.0" encoding="UTF-8"?>
<jaxws-config xmlns="urn:jboss:jbossws-jaxws-config:4.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:javaee="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="urn:jboss:jbossws-jaxws-config:4.0 schema/jbossws-jaxws-config_4_0.xsd">
<endpoint-config>
<config-name>org.jboss.test.ws.jaxws.cxf.interceptors.EndpointImpl</config-name>
<property>
<property-name>cxf.interceptors.in</property-name>
<property-value>org.jboss.test.ws.jaxws.cxf.interceptors.EndpointInterceptor,org.jboss.test.ws.jax
```

A new instance of each specified interceptor class will be added to the client or endpoint the configuration is assigned to. The interceptor classes must have a no-argument constructor.

Apache CXF features

Apache CXF supports declaring features using one of the following approaches:

- Annotation usage on endpoint classes (`@org.apache.cxf.feature.Features`)
- Direct API usage on client side (through extensions of the `org.apache.cxf.feature.AbstractFeature` class)
- Spring descriptor usage (*cxf.xml*)

As the Spring descriptor usage is not supported, the JBossWS integration adds an additional descriptor based approach to avoid requiring modifications to the actual client/endpoint code. Users can declare features within predefined client and endpoint configurations by specifying a list of feature class names for the `cxf.features` property.

```
<?xml version="1.0" encoding="UTF-8"?>
<jaxws-config xmlns="urn:jboss:jbossws-jaxws-config:4.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:javaee="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="urn:jboss:jbossws-jaxws-config:4.0 schema/jbossws-jaxws-config_4_0.xsd">
<endpoint-config>
<config-name>Custom FI Config</config-name>
<property>
<property-name>cxf.features</property-name>
<property-value>org.apache.cxf.feature.FastInfosetFeature</property-value>
</property>
</endpoint-config>
</jaxws-config>
```

A new instance of each specified feature class will be added to the client or endpoint the configuration is assigned to. The feature classes must have a no-argument constructor.

Properties driven bean creation

Sections above explain how to declare CXF interceptors and features through properties either in a client/endpoint predefined configuration or in a `jboss-webservices.xml` descriptor. By getting the feature/interceptor class name only specified, the container simply tries to create a bean instance using the class default constructor. This sets a limitation on the feature/interceptor configuration, unless custom extensions of vanilla CXF classes are provided, with the default constructor setting properties before eventually using the super constructor.

To cope with this issue, JBossWS integration comes with a mechanism for configuring simple bean hierarchies when building them up from properties. Properties can have bean reference values, that is strings starting with ##. Property reference keys are used to specify the bean class name and the value for for each attribute. So for instance the following properties:

Key	Value
cxf.features	##foo, ##bar
##foo	org.jboss.Foo
##foo.par	34
##bar	org.jboss.Bar
##bar.color	blue

would result into the stack installing two feature instances, the same that would have been created by

```
import org.Bar;
import org.Foo;

...

Foo foo = new Foo();
foo.setPar(34);
Bar bar = new Bar();
bar.setColor("blue");
```

The mechanism assumes that the classes are valid beans with proper getter and setter methods; value objects are cast to the correct primitive type by inspecting the class definition. Nested beans can of course be configured.

HTTPConduit configuration

HTTP transport setup in Apache CXF is achieved through `org.apache.cxf.transport.http.HTTPConduit` configurations. When running on top of the JBossWS integration, conduits can be programmatically modified using the Apache CXF API as follows:

```
import org.apache.cxf.frontend.ClientProxy;
import org.apache.cxf.transport.http.HTTPConduit;
import org.apache.cxf.transports.http.configuration.HTTPClientPolicy;

//set chunking threshold before using a JAX-WS port client
...
HTTPConduit conduit = (HTTPConduit)ClientProxy.getClient(port).getConduit();
HTTPClientPolicy client = conduit.getClient();

client.setChunkingThreshold(8192);
...
```

Users can also control the default values for the most common HTTPConduit parameters by setting specific system properties; the provided values will override Apache CXF defaut values.

Property	Description
cxf.client.allowChunking	A boolean to tell Apache CXF whether to allow send messages using chunking.
cxf.client.chunkingThreshold	An integer value to tell Apache CXF the threshold at which switching from non-chunking to chunking mode.
cxf.client.connectionTimeout	A long value to tell Apache CXF how many milliseconds to set the connection timeout to
cxf.client.receiveTimeout	A long value to tell Apache CXF how many milliseconds to set the receive timeout to
cxf.client.connection	A string to tell Apache CXF to use `Keep-Alive` or `close` connection type
cxf.tls-client.disableCNCheck	A boolean to tell Apache CXF whether disabling CN host name check or not

The vanilla Apache CXF defaults apply when the system properties above are not set.

Addressing

JBoss Web Services inherits full WS-Addressing capabilities from the underlying Apache CXF implementation. Apache CXF provides support for 2004-08 and 1.0 versions of WS-Addressing.

Enabling WS-Addressing

WS-Addressing can be turned on in multiple standard ways:

- consuming a WSDL contract that specifies a WS-Addressing assertion / policy
- using the `@javax.xml.ws.soap.Addressing` annotation
- using the `javax.xml.ws.soap.AddressingFeature` feature

The supported addressing policy elements are:

```
[http://www.w3.org/2005/02/addressing/wsdl]UsingAddressing
[http://schemas.xmlsoap.org/ws/2004/08/addressing/policy]UsingAddressing
[http://www.w3.org/2006/05/addressing/wsdl]UsingAddressing
[http://www.w3.org/2007/05/addressing/metadata]Addressing
```

Alternatively, Apache CXF proprietary ways are also available:

- specifying the *[http://cxf.apache.org/ws/addressing]addressing* feature for a given client/endpoint
- using the `org.apache.cxf.ws.addressing.WSAddressingFeature` feature through the API
- manually configuring the Apache CXF addressing interceptors (`org.apache.cxf.ws.addressing.MAPAggregator` and `org.apache.cxf.ws.addressing.soap.MAPCodec`)
- setting the *org.apache.cxf.ws.addressing.using* property in the message context

Please refer to the the Apache CXF documentation for further information on the proprietary WS-Addressing setup and configuration details.

Addressing Policy

The WS-Addressing support is also perfectly integrated with the Apache CXF WS-Policy engine.

This basically means that the WSDL contract generation for code-first endpoint deployment is policy-aware: users can annotate endpoints with the `@javax.xml.ws.soap.Addressing` annotation and expect the published WSDL contract to contain proper WS-Addressing policy (assuming no `wsdlLocation` is specified in the endpoint's `@WebService` annotation).

Similarly, on client side users do not need to manually specify the `javax.xml.ws.soap.AddressingFeature` feature, as the policy engine is able to properly process the WS-Addressing policy in the consumed WSDL and turn on addressing as requested.

Example

Here is an example showing how to simply enable WS-Addressing through WS-Policy.

Endpoint

A simple JAX-WS endpoint is prepared using a java-first approach; WS-Addressing is enforced through `@Addressing` annotation and no `wsdlLocation` is provided in `@WebService`:

```
package org.jboss.test.ws.jaxws.samples.wsa;

import javax.jws.WebService;
import javax.xml.ws.soap.Addressing;
import org.jboss.logging.Logger;

@WebService
(
   portName = "AddressingServicePort",
   serviceName = "AddressingService",
   targetNamespace = "http://www.jboss.org/jbossws/ws-extensions/wsaddressing",
   endpointInterface = "org.jboss.test.ws.jaxws.samples.wsa.ServiceIface"
)
@Addressing(enabled=true, required=true)
public class ServiceImpl implements ServiceIface
{
   private Logger log = Logger.getLogger(this.getClass());

   public String sayHello(String name)
   {
      return "Hello " + name + "!";
   }
}
```

The WSDL contract that's generated at deploy time and published looks like this:

```
<wsdl:definitions ....>
...
  <wsdl:binding name="AddressingServiceSoapBinding" type="tns:ServiceIface">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsaw:UsingAddressing wsdl:required="true"/>
    <wsp:PolicyReference URI="#AddressingServiceSoapBinding_WSAM_Addressing_Policy"/>

    <wsdl:operation name="sayHello">
      <soap:operation soapAction="" style="document"/>
      <wsdl:input name="sayHello">
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output name="sayHelloResponse">
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>

  </wsdl:binding>
  <wsdl:service name="AddressingService">
    <wsdl:port binding="tns:AddressingServiceSoapBinding" name="AddressingServicePort">
      <soap:address location="http://localhost:8080/jaxws-samples-wsa"/>
    </wsdl:port>
  </wsdl:service>
    <wsp:Policy wsu:Id="AddressingServiceSoapBinding_WSAM_Addressing_Policy"

xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">
      <wsam:Addressing xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata">
        <wsp:Policy/>
      </wsam:Addressing>
    </wsp:Policy>
</wsdl:definitions>
```

Client

Since the WS-Policy engine is on by default, the client side code is basically a pure JAX-WS client app:

```
QName serviceName = new QName("http://www.jboss.org/jbossws/ws-extensions/wsaddressing",
"AddressingService");
URL wsdlURL = new URL("http://localhost:8080/jaxws-samples-wsa?wsdl");
Service service = Service.create(wsdlURL, serviceName);
ServiceIface proxy = (ServiceIface)service.getPort(ServiceIface.class);
proxy.sayHello("World");
```

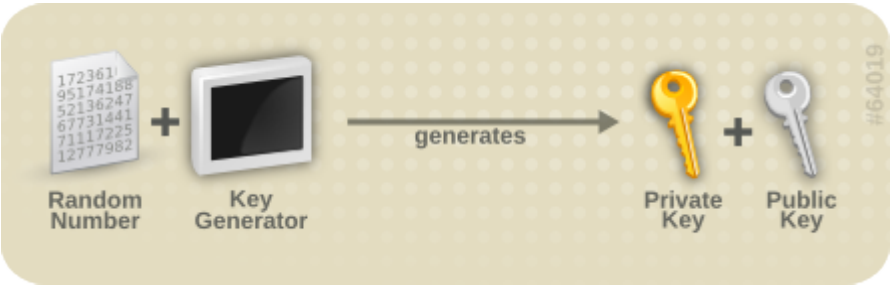
Security

Security overview

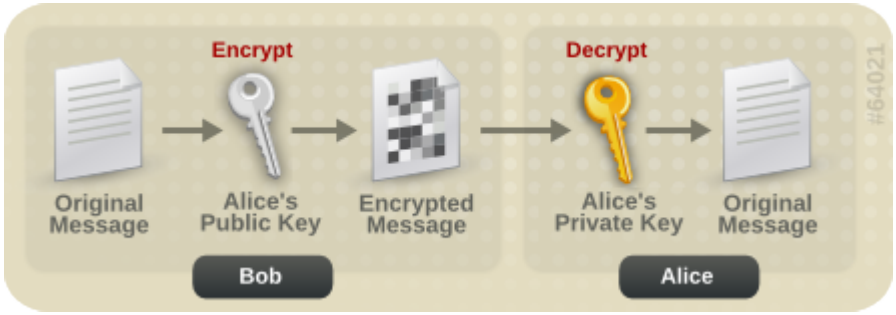
WS-Security provides the means to secure your services beyond transport level protocols such as *HTTPS*. Through a number of standards such as XML-Encryption, and headers defined in the WS-Security standard, it allows you to:

- Pass authentication tokens between services.
- Encrypt messages or parts of messages.
- Sign messages.
- Timestamp messages.

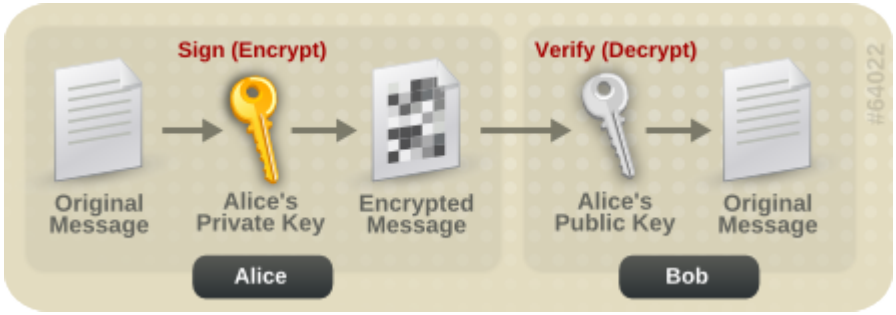
WS-Security makes heavy use of public and private key cryptography. It is helpful to understand these basics to really understand how to configure WS-Security. With public key cryptography, a user has a pair of public and private keys. These are generated using a large prime number and a key function.



The keys are related mathematically, but cannot be derived from one another. With these keys we can encrypt messages. For example, if Bob wants to send a message to Alice, he can encrypt a message using her public key. Alice can then decrypt this message using her private key. Only Alice can decrypt this message as she is the only one with the private key.



Messages can also be signed. This allows you to ensure the authenticity of the message. If Alice wants to send a message to Bob, and Bob wants to be sure that it is from Alice, Alice can sign the message using her private key. Bob can then verify that the message is from Alice by using her public key.



JBoss WS-Security support

JBoss Web Services supports many real world scenarios requiring WS-Security functionalities. This includes signature and encryption support through X509 certificates, authentication and authorization through username tokens as well as all ws-security configurations covered by WS-SecurityPolicy specification.

As well as for other WS-* features, the core of WS-Security functionalities is provided through the Apache CXF engine. On top of that the JBossWS integration adds few configuration enhancements to simplify the setup of WS-Security enabled endpoints.

Apache CXF WS-Security implementation

Apache CXF features a top class WS-Security module supporting multiple configurations and easily extendible.

The system is based on *interceptors* that delegate to Apache WSS4J for the low level security operations. Interceptors can be configured in different ways, either through Spring configuration files or directly using Apache CXF client API. Please refer to the Apache CXF documentation if you're looking for more details.

Recent versions of Apache CXF, however, introduced support for WS-Security Policy, which aims at moving most of the security configuration into the service contract (through policies), so that clients can easily be configured almost completely automatically from that. This way users do not need to manually deal with configuring / installing the required interceptors; the Apache CXF WS-Policy engine internally takes care of that instead.

Security Policy support

WS-SecurityPolicy describes the actions that are required to securely communicate with a service advertised in a given WSDL contract. The WSDL bindings / operations reference WS-Policy fragments with the security requirements to interact with the service. The WS-SecurityPolicy specification allows for specifying things like asymmetric/symmetric keys, using transports (https) for encryption, which parts/headers to encrypt or sign, whether to sign then encrypt or encrypt then sign, whether to include timestamps, whether to use derived keys, etc.

However some mandatory configuration elements are not covered by WS-SecurityPolicy, basically because they're not meant to be public / part of the published endpoint contract; those include things such as keystore locations, usernames and passwords, etc. Apache CXF allows configuring these elements either through Spring xml descriptors or using the client API / annotations. Below is the list of supported configuration properties:

ws-security.username	The username used for UsernameToken policy assertions
ws-security.password	The password used for UsernameToken policy assertions. If not specified, the callback handler will be called.
ws-security.callback-handler	The WSS4J security CallbackHandler that will be used to retrieve passwords for keystores and UsernameTokens.
ws-security.signature.properties	The properties file/object that contains the WSS4J properties for configuring the signature keystore and crypto objects
ws-security.encryption.properties	The properties file/object that contains the WSS4J properties for configuring the encryption keystore and crypto objects
ws-security.signature.username	The username or alias for the key in the signature keystore that will be used. If not specified, it uses the the default alias set in the properties file. If that's also not set, and the keystore only contains a single key, that key will be used.
ws-security.encryption.username	The username or alias for the key in the encryption keystore that will be used. If not specified, it uses the the default alias set in the properties file. If that's also not set, and the keystore only contains a single key, that key will be used. For the web service provider, the useReqSigCert keyword can be used to accept (encrypt to) any client whose public key is in the service's truststore (defined in ws-security.encryption.properties.)
ws-security.signature.crypto	Instead of specifying the signature properties, this can point to the full WSS4J Crypto object. This can allow easier "programmatic" configuration of the Crypto information."
ws-security.encryption.crypto	Instead of specifying the encryption properties, this can point to the full WSS4J Crypto object. This can allow easier "programmatic" configuration of the Crypto information."
ws-security.enable.streaming	Enable streaming (StAX based) processing of WS-Security messages

Here is an example of configuration using the client API:

```
Map<String, Object> ctx = ((BindingProvider)port).getRequestContext();
ctx.put("ws-security.encryption.properties", properties);
port.echoString("hello");
```

Please refer to the Apache CXF documentation for additional configuration details.

JBossWS configuration additions

In order for removing the need of Spring on server side for setting up WS-Security configuration properties not covered by policies, the JBossWS integration allows for getting those pieces of information from a defined *endpoint configuration*. Endpoint configurations can include property declarations and endpoint implementations can be associated with a given endpoint configuration using the `@EndpointConfig` annotation.

```
<?xml version="1.0" encoding="UTF-8"?>
<jaxws-config xmlns="urn:jboss:jbossws-jaxws-config:4.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:javaee="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="urn:jboss:jbossws-jaxws-config:4.0 schema/jbossws-jaxws-config_4_0.xsd">
  <endpoint-config>
    <config-name>Custom WS-Security Endpoint</config-name>
    <property>
      <property-name>ws-security.signature.properties</property-name>
      <property-value>bob.properties</property-value>
    </property>
    <property>
      <property-name>ws-security.encryption.properties</property-name>
      <property-value>bob.properties</property-value>
    </property>
    <property>
      <property-name>ws-security.signature.username</property-name>
      <property-value>bob</property-value>
    </property>
    <property>
      <property-name>ws-security.encryption.username</property-name>
      <property-value>alice</property-value>
    </property>
    <property>
      <property-name>ws-security.callback-handler</property-name>

<property-value>org.jboss.test.ws.jaxws.samples.wsse.policy.basic.KeystorePasswordCallback</proper
</property>
  </endpoint-config>
</jaxws-config>
```

```
import javax.jws.WebService;
import org.jboss.ws.api.annotation.EndpointConfig;

@WebService
(
   portName = "SecurityServicePort",
   serviceName = "SecurityService",
   wsdlLocation = "WEB-INF/wsdl/SecurityService.wsdl",
   targetNamespace = "http://www.jboss.org/jbossws/ws-extensions/wssecuritypolicy",
   endpointInterface = "org.jboss.test.ws.jaxws.samples.wsse.policy.basic.ServiceIface"
)
@EndpointConfig(configFile = "WEB-INF/jaxws-endpoint-config.xml", configName = "Custom
WS-Security Endpoint")
public class ServiceImpl implements ServiceIface
{
   public String sayHello()
   {
      return "Secure Hello World!";
   }
}
```

Apache CXF annotations

The JBossWS configuration additions allow for a descriptor approach to the WS-Security Policy engine configuration. If you prefer to provide the same information through an annotation approach, you can leverage the Apache CXF `@org.apache.cxf.annotations.EndpointProperties` annotation:

```
@WebService(
   ...
)
@EndpointProperties(value = {
      @EndpointProperty(key = "ws-security.signature.properties", value = "bob.properties"),
      @EndpointProperty(key = "ws-security.encryption.properties", value = "bob.properties"),
      @EndpointProperty(key = "ws-security.signature.username", value = "bob"),
      @EndpointProperty(key = "ws-security.encryption.username", value = "alice"),
      @EndpointProperty(key = "ws-security.callback-handler", value =
"org.jboss.test.ws.jaxws.samples.wsse.policy.basic.KeystorePasswordCallback")
      }
)
public class ServiceImpl implements ServiceIface {
   ...
}
```

Examples

In this section some sample of WS-Security service endpoints and clients are provided. Please note they're only meant as tutorials; you should really careful isolate the ws-security policies / assertion that best suite your security needs before going to production environment.

The following sections provide directions and examples on understanding some of the configuration options for WS-Security engine. Please note the implementor remains responsible for assessing the application requirements and choosing the most suitable security policy for them.

Signature and encryption

Endpoint

First of all you need to create the web service endpoint using JAX-WS. While this can generally be achieved in different ways, it's required to use a contract-first approach when using WS-Security, as the policies declared in the wsdl are parsed by the Apache CXF engine on both server and client sides. So, here is an example of WSDL contract enforcing signature and encryption using X 509 certificates (the referenced schema is omitted):

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions targetNamespace="http://www.jboss.org/jbossws/ws-extensions/wssecuritypolicy"
name="SecurityService"
  xmlns:tns="http://www.jboss.org/jbossws/ws-extensions/wssecuritypolicy"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:wsp="http://www.w3.org/ns/ws-policy"

xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
        xmlns:wsaws="http://www.w3.org/2005/08/addressing"
        xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://www.jboss.org/jbossws/ws-extensions/wssecuritypolicy"
schemaLocation="SecurityService_schema1.xsd"/>
    </xsd:schema>
  </types>
  <message name="sayHello">
    <part name="parameters" element="tns:sayHello"/>
  </message>
  <message name="sayHelloResponse">
    <part name="parameters" element="tns:sayHelloResponse"/>
  </message>
  <portType name="ServiceIface">
    <operation name="sayHello">
      <input message="tns:sayHello"/>
      <output message="tns:sayHelloResponse"/>
    </operation>
  </portType>
  <binding name="SecurityServicePortBinding" type="tns:ServiceIface">
    <wsp:PolicyReference URI="#SecurityServiceSignThenEncryptPolicy"/>
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
    <operation name="sayHello">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
```

```
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>
  <service name="SecurityService">
    <port name="SecurityServicePort" binding="tns:SecurityServicePortBinding">
      <soap:address location="http://localhost:8080/jaxws-samples-wssePolicy-sign-encrypt"/>
    </port>
  </service>

  <wsp:Policy wsu:Id="SecurityServiceSignThenEncryptPolicy"
xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
    <wsp:ExactlyOne>
      <wsp:All>
        <sp:AsymmetricBinding xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
          <wsp:Policy>
            <sp:InitiatorToken>
              <wsp:Policy>
                <sp:X509Token
sp:IncludeToken="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy/IncludeToken/AlwaysToRecipie
<wsp:Policy>
                    <sp:WssX509V1Token11/>
                  </wsp:Policy>
                  </sp:X509Token>
              </wsp:Policy>
            </sp:InitiatorToken>
            <sp:RecipientToken>
              <wsp:Policy>
                <sp:X509Token
sp:IncludeToken="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy/IncludeToken/Never">
                  <wsp:Policy>
                    <sp:WssX509V1Token11/>
                  </wsp:Policy>
                </sp:X509Token>
              </wsp:Policy>
            </sp:RecipientToken>
            <sp:AlgorithmSuite>
              <wsp:Policy>
                <sp:TripleDesRsa15/>
              </wsp:Policy>
            </sp:AlgorithmSuite>
            <sp:Layout>
              <wsp:Policy>
                <sp:Lax/>
              </wsp:Policy>
            </sp:Layout>
            <sp:IncludeTimestamp/>
            <sp:EncryptSignature/>
            <sp:OnlySignEntireHeadersAndBody/>
            <sp:SignBeforeEncrypting/>
          </wsp:Policy>
        </sp:AsymmetricBinding>
        <sp:SignedParts xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
          <sp:Body/>
        </sp:SignedParts>
        <sp:EncryptedParts xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
          <sp:Body/>
        </sp:EncryptedParts>
```

```
        <sp:Wss10 xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
          <wsp:Policy>
            <sp:MustSupportRefIssuerSerial/>
          </wsp:Policy>
        </sp:Wss10>
      </wsp:All>
    </wsp:ExactlyOne>
  </wsp:Policy>
</definitions>
```

The service endpoint can be generated using the `wsconsume` tool and then enriched with a `@EndpointConfig` annotation:

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.basic;

import javax.jws.WebService;
import org.jboss.ws.api.annotation.EndpointConfig;

@WebService
(
   portName = "SecurityServicePort",
   serviceName = "SecurityService",
   wsdlLocation = "WEB-INF/wsdl/SecurityService.wsdl",
   targetNamespace = "http://www.jboss.org/jbossws/ws-extensions/wssecuritypolicy",
   endpointInterface = "org.jboss.test.ws.jaxws.samples.wsse.policy.basic.ServiceIface"
)
@EndpointConfig(configFile = "WEB-INF/jaxws-endpoint-config.xml", configName = "Custom
WS-Security Endpoint")
public class ServiceImpl implements ServiceIface
{
   public String sayHello()
   {
      return "Secure Hello World!";
   }
}
```

The referenced *jaxws-endpoint-config.xml* descriptor is used to provide a custom endpoint configuration with the required server side configuration properties; this tells the engine which certificate / key to use for signature / signature verification and for encryption / decryption:

```
<?xml version="1.0" encoding="UTF-8"?>
<jaxws-config xmlns="urn:jboss:jbossws-jaxws-config:4.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:javaee="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="urn:jboss:jbossws-jaxws-config:4.0 schema/jbossws-jaxws-config_4_0.xsd">
  <endpoint-config>
    <config-name>Custom WS-Security Endpoint</config-name>
    <property>
      <property-name>ws-security.signature.properties</property-name>
      <property-value>bob.properties</property-value>
    </property>
    <property>
      <property-name>ws-security.encryption.properties</property-name>
      <property-value>bob.properties</property-value>
    </property>
    <property>
      <property-name>ws-security.signature.username</property-name>
      <property-value>bob</property-value>
    </property>
    <property>
      <property-name>ws-security.encryption.username</property-name>
      <property-value>alice</property-value>
    </property>
    <property>
      <property-name>ws-security.callback-handler</property-name>

<property-value>org.jboss.test.ws.jaxws.samples.wsse.policy.basic.KeystorePasswordCallback</proper
</property>
  </endpoint-config>
</jaxws-config>
```

... the *bob.properties* configuration file is also referenced above; it includes the WSS4J Crypto properties which in turn link to the keystore file, type and the alias/password to use for accessing it:

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=password
org.apache.ws.security.crypto.merlin.keystore.alias=bob
org.apache.ws.security.crypto.merlin.keystore.file=bob.jks
```

A callback handler for the letting Apache CXF access the keystore is also provided:

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.basic;

import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import org.apache.ws.security.WSPasswordCallback;

public class KeystorePasswordCallback implements CallbackHandler {
   private Map<String, String> passwords = new HashMap<String, String>();

   public KeystorePasswordCallback() {
      passwords.put("alice", "password");
      passwords.put("bob", "password");
   }

   /**
    * It attempts to get the password from the private
    * alias/passwords map.
    */
   public void handle(Callback[] callbacks) throws IOException, UnsupportedCallbackException {
      for (int i = 0; i < callbacks.length; i++) {
         WSPasswordCallback pc = (WSPasswordCallback)callbacks[i];

         String pass = passwords.get(pc.getIdentifier());
         if (pass != null) {
            pc.setPassword(pass);
            return;
         }
      }
   }

   /**
    * Add an alias/password pair to the callback mechanism.
    */
   public void setAliasPassword(String alias, String password) {
      passwords.put(alias, password);
   }
}
```

Assuming the *bob.jks* keystore has been properly generated and contains Bob's (server) full key (private/certificate + public key) as well as Alice's (client) public key, we can proceed to packaging the endpoint. Here is the expected content (the endpoint is a *POJO* one in a *war* archive, but *EJB3* endpoints in *jar* archives are of course also supported):

```
alessio@inuyasha /dati/jbossws/stack/cxf/trunk $ jar -tvf
./modules/testsuite/cxf-tests/target/test-libs/jaxws-samples-wsse-policy-sign-encrypt.war
     0 Thu Jun 16 18:50:48 CEST 2011 META-INF/
   140 Thu Jun 16 18:50:46 CEST 2011 META-INF/MANIFEST.MF
     0 Thu Jun 16 18:50:48 CEST 2011 WEB-INF/
   586 Thu Jun 16 18:50:44 CEST 2011 WEB-INF/web.xml
     0 Thu Jun 16 18:50:48 CEST 2011 WEB-INF/classes/
     0 Thu Jun 16 18:50:48 CEST 2011 WEB-INF/classes/org/
     0 Thu Jun 16 18:50:48 CEST 2011 WEB-INF/classes/org/jboss/
     0 Thu Jun 16 18:50:48 CEST 2011 WEB-INF/classes/org/jboss/test/
     0 Thu Jun 16 18:50:48 CEST 2011 WEB-INF/classes/org/jboss/test/ws/
     0 Thu Jun 16 18:50:48 CEST 2011 WEB-INF/classes/org/jboss/test/ws/jaxws/
     0 Thu Jun 16 18:50:48 CEST 2011 WEB-INF/classes/org/jboss/test/ws/jaxws/samples/
     0 Thu Jun 16 18:50:48 CEST 2011 WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/
     0 Thu Jun 16 18:50:48 CEST 2011
WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/
     0 Thu Jun 16 18:50:48 CEST 2011
WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/basic/
  1687 Thu Jun 16 18:50:48 CEST 2011
WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/basic/KeystorePasswordCallback.class
   383 Thu Jun 16 18:50:48 CEST 2011
WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/basic/ServiceIface.class
  1070 Thu Jun 16 18:50:48 CEST 2011
WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/basic/ServiceImpl.class
     0 Thu Jun 16 18:50:48 CEST 2011
WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/jaxws/
   705 Thu Jun 16 18:50:48 CEST 2011
WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/jaxws/SayHello.class
  1069 Thu Jun 16 18:50:48 CEST 2011
WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/jaxws/SayHelloResponse.class
  1225 Thu Jun 16 18:50:44 CEST 2011 WEB-INF/jaxws-endpoint-config.xml
     0 Thu Jun 16 18:50:44 CEST 2011 WEB-INF/wsdl/
  4086 Thu Jun 16 18:50:44 CEST 2011 WEB-INF/wsdl/SecurityService.wsdl
   653 Thu Jun 16 18:50:44 CEST 2011 WEB-INF/wsdl/SecurityService_schema1.xsd
  1820 Thu Jun 16 18:50:44 CEST 2011 WEB-INF/classes/bob.jks
   311 Thu Jun 16 18:50:44 CEST 2011 WEB-INF/classes/bob.properties
```

As you can see, the jaxws classes generated by the tools are of course also included, as well as a basic *web.xml* referencing the endpoint bean:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app
   version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
   <servlet>
      <servlet-name>TestService</servlet-name>

<servlet-class>org.jboss.test.ws.jaxws.samples.wsse.policy.basic.ServiceImpl</servlet-class>
   </servlet>
   <servlet-mapping>
      <servlet-name>TestService</servlet-name>
      <url-pattern>/*</url-pattern>
   </servlet-mapping>
</web-app>
```

If you're deploying the endpoint archive on WildFly, remember to add a dependency to *org.apache.ws.security* module in the MANIFEST.MF file.

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.7.1
Created-By: 17.0-b16 (Sun Microsystems Inc.)
Dependencies: org.apache.ws.security
```

Client

You start by consuming the published WSDL contract using the *wsconsume* tool on client side too. Then you simply invoke the the endpoint as a standard JAX-WS one:

```
QName serviceName = new QName("http://www.jboss.org/jbossws/ws-extensions/wssecuritypolicy",
"SecurityService");
URL wsdlURL = new URL(serviceURL + "?wsdl");
Service service = Service.create(wsdlURL, serviceName);
ServiceIface proxy = (ServiceIface)service.getPort(ServiceIface.class);

((BindingProvider)proxy).getRequestContext().put(SecurityConstants.CALLBACK_HANDLER, new
KeystorePasswordCallback());
((BindingProvider)proxy).getRequestContext().put(SecurityConstants.SIGNATURE_PROPERTIES,
     Thread.currentThread().getContextClassLoader().getResource("META-INF/alice.properties"));
((BindingProvider)proxy).getRequestContext().put(SecurityConstants.ENCRYPT_PROPERTIES,
     Thread.currentThread().getContextClassLoader().getResource("META-INF/alice.properties"));
((BindingProvider)proxy).getRequestContext().put(SecurityConstants.SIGNATURE_USERNAME, "alice");
((BindingProvider)proxy).getRequestContext().put(SecurityConstants.ENCRYPT_USERNAME, "bob");

proxy.sayHello();
```

As you can see, the WS-Security properties are set in the request context. Here the `KeystorePasswordCallback` is the same as on server side above, you might want/need different implementation in real world scenarios, of course.

The *alice.properties* file is the client side equivalent of the server side *bob.properties* and references the *alice.jks* keystore file, which has been populated with Alice's (client) full key (private/certificate + public key) as well as Bob's (server) public key.

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=password
org.apache.ws.security.crypto.merlin.keystore.alias=alice
org.apache.ws.security.crypto.merlin.keystore.file=META-INF/alice.jks
```

The Apache CXF WS-Policy engine will digest the security requirements in the contract and ensure a valid secure communication is in place for interacting with the server endpoint.

Endpoint serving multiple clients

The server side configuration described above implies the endpoint is configured for serving a given client which a service agreement has been established for. In some real world scenarios though, the same server might be expected to be able to deal with (including decrypting and encrypting) messages coming from and being sent to multiple clients. Apache CXF supports that through the `useReqSigCert` value for the `ws-security.encryption.username` configuration parameter.

Of course the referenced server side keystore then needs to contain the public key of all the clients that are expected to be served.

Authentication and authorization

The Username Token Profile can be used to provide client's credentials to a WS-Security enabled target endpoint.

Apache CXF provides means for setting basic *password callback handlers* on both client and server sides to set/check passwords; the *ws-security.username* and *ws-security.callback-handler* properties can be used similarly as shown in the signature and encryption example. Things become more interesting when requiring a given user to be authenticated (and authorized) against a security domain on the target application server.

On server side, you need to install two additional interceptors that act as bridges towards the application server authentication layer:

- an interceptor for performing authentication and populating a valid SecurityContext; the provided interceptor should extend org.apache.cxf.ws.interceptor.security.AbstractUsernameTokenInInterceptor, in particular JBossWS integration comes with *org.jboss.wsf.stack.cxf.security.authentication.SubjectCreatingInterceptor* for this;
- an interceptor for performing authorization; CXF requires that to extend org.apache.cxf.interceptor.security.AbstractAuthorizingInInterceptor, for instance the *SimpleAuthorizingInterceptor* can be used for simply mapping endpoint operations to allowed roles.

So, here follows an example of WS-SecurityPolicy endpoint using Username Token Profile for authenticating through the application server security domain system.

Endpoint
As in the other example, we start with a wsdl contract containing the proper WS-Security Policy:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions targetNamespace="http://www.jboss.org/jbossws/ws-extensions/wssecuritypolicy"
name="SecurityService"
  xmlns:tns="http://www.jboss.org/jbossws/ws-extensions/wssecuritypolicy"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"

xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
        xmlns:wsaws="http://www.w3.org/2005/08/addressing">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://www.jboss.org/jbossws/ws-extensions/wssecuritypolicy"
schemaLocation="SecurityService_schema1.xsd"/>
    </xsd:schema>
  </types>
  <message name="sayHello">
    <part name="parameters" element="tns:sayHello"/>
  </message>
  <message name="sayHelloResponse">
    <part name="parameters" element="tns:sayHelloResponse"/>
  </message>
  <message name="greetMe">
    <part name="parameters" element="tns:greetMe"/>
  </message>
  <message name="greetMeResponse">
    <part name="parameters" element="tns:greetMeResponse"/>
  </message>
  <portType name="ServiceIface">
```

```
    <operation name="sayHello">
      <input message="tns:sayHello"/>
      <output message="tns:sayHelloResponse"/>
    </operation>
    <operation name="greetMe">
      <input message="tns:greetMe"/>
      <output message="tns:greetMeResponse"/>
    </operation>
  </portType>
  <binding name="SecurityServicePortBinding" type="tns:ServiceIface">
    <wsp:PolicyReference URI="#SecurityServiceUsernameUnsecureTransportPolicy"/>
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
    <operation name="sayHello">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
    <operation name="greetMe">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>
  <service name="SecurityService">
    <port name="SecurityServicePort" binding="tns:SecurityServicePortBinding">
      <soap:address location="http://localhost:8080/jaxws-samples-wsse-username-jaas"/>
    </port>
  </service>

  <wsp:Policy wsu:Id="SecurityServiceUsernameUnsecureTransportPolicy">
        <wsp:ExactlyOne>
            <wsp:All>
                <sp:SupportingTokens
xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
                    <wsp:Policy>
                        <sp:UsernameToken
sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/AlwaysToRe
<wsp:Policy>
                                <sp:WssUsernameToken10/>
                            </wsp:Policy>
                        </sp:UsernameToken>
                    </wsp:Policy>
                </sp:SupportingTokens>
            </wsp:All>
        </wsp:ExactlyOne>
    </wsp:Policy>

</definitions>
```

If you want to send hash / digest passwords, you can use a policy such as what follows:

```
<wsp:Policy wsu:Id="SecurityServiceUsernameHashPasswordPolicy">
    <wsp:ExactlyOne>
        <wsp:All>
            <sp:SupportingTokens
xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
                <wsp:Policy>
                    <sp:UsernameToken
sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/Alwa
<wsp:Policy>
                            <sp:HashPassword/>
                        </wsp:Policy>
                    </sp:UsernameToken>
                </wsp:Policy>
            </sp:SupportingTokens>
        </wsp:All>
    </wsp:ExactlyOne>
</wsp:Policy>
```

Please note the specified JBoss security domain needs to be properly configured for computing digests.

The service endpoint can be generated using the `wsconsume` tool and then enriched with a `@EndpointConfig` annotation and `@InInterceptors` annotation to add the two interceptors mentioned above for JAAS integration:

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.jaas;

import javax.jws.WebService;
import org.apache.cxf.interceptor.InInterceptors;
import org.jboss.ws.api.annotation.EndpointConfig;

@WebService
(
   portName = "SecurityServicePort",
   serviceName = "SecurityService",
   wsdlLocation = "WEB-INF/wsdl/SecurityService.wsdl",
   targetNamespace = "http://www.jboss.org/jbossws/ws-extensions/wssecuritypolicy",
   endpointInterface = "org.jboss.test.ws.jaxws.samples.wsse.policy.jaas.ServiceIface"
)
@EndpointConfig(configFile = "WEB-INF/jaxws-endpoint-config.xml", configName = "Custom
WS-Security Endpoint")
@InInterceptors(interceptors = {
      "org.jboss.wsf.stack.cxf.security.authentication.SubjectCreatingPolicyInterceptor",
      "org.jboss.test.ws.jaxws.samples.wsse.policy.jaas.POJOEndpointAuthorizationInterceptor"}
)
public class ServiceImpl implements ServiceIface
{
   public String sayHello()
   {
      return "Secure Hello World!";
   }

   public String greetMe()
   {
      return "Greetings!";
   }
}
```

The `POJOEndpointAuthorizationInterceptor` is included into the deployment and deals with the roles cheks:

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.jaas;

import java.util.HashMap;
import java.util.Map;
import org.apache.cxf.interceptor.security.SimpleAuthorizingInterceptor;

public class POJOEndpointAuthorizationInterceptor extends SimpleAuthorizingInterceptor
{

   public POJOEndpointAuthorizationInterceptor()
   {
      super();
      readRoles();
   }

   private void readRoles()
   {
      //just an example, this might read from a configuration file or such
      Map<String, String> roles = new HashMap<String, String>();
      roles.put("sayHello", "friend");
      roles.put("greetMe", "snoopies");
      setMethodRolesMap(roles);
   }
}
```

The *jaxws-endpoint-config.xml* descriptor is used to provide a custom endpoint configuration with the required server side configuration properties; in particular for this Username Token case that's just a CXF configuration option for leaving the username token validation to the configured interceptors:

```
<?xml version="1.0" encoding="UTF-8"?>
<jaxws-config xmlns="urn:jboss:jbossws-jaxws-config:4.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:javaee="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="urn:jboss:jbossws-jaxws-config:4.0 schema/jbossws-jaxws-config_4_0.xsd">
  <endpoint-config>
    <config-name>Custom WS-Security Endpoint</config-name>
    <property>
      <property-name>ws-security.validate.token</property-name>
      <property-value>false</property-value>
    </property>
  </endpoint-config>
</jaxws-config>
```

In order for requiring a given JBoss security domain to be used to protect access to the endpoint (a POJO one in this case), we declare that in a *jboss-web.xml* descriptor (the *JBossWS* security domain is used):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jboss-web PUBLIC "-//JBoss//DTD Web Application 2.4//EN"
"http://www.jboss.org/j2ee/dtd/jboss-web_4_0.dtd">
<jboss-web>
   <security-domain>java:/jaas/JBossWS</security-domain>
</jboss-web
```

Finally, the *web.xml* is as simple as usual:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app
   version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
   <servlet>
      <servlet-name>TestService</servlet-name>

<servlet-class>org.jboss.test.ws.jaxws.samples.wsse.policy.jaas.ServiceImpl</servlet-class>
   </servlet>
   <servlet-mapping>
      <servlet-name>TestService</servlet-name>
      <url-pattern>/*</url-pattern>
   </servlet-mapping>
</web-app>
```

The endpoint is packaged into a war archive, including the JAXWS classes generated by wsconsume:

```
alessio@inuyasha /dati/jbossws/stack/cxf/trunk $ jar -tvf
./modules/testsuite/cxf-tests/target/test-libs/jaxws-samples-wsse-policy-username-jaas.war
     0 Thu Jun 16 18:50:48 CEST 2011 META-INF/
   155 Thu Jun 16 18:50:46 CEST 2011 META-INF/MANIFEST.MF
     0 Thu Jun 16 18:50:48 CEST 2011 WEB-INF/
   585 Thu Jun 16 18:50:44 CEST 2011 WEB-INF/web.xml
     0 Thu Jun 16 18:50:48 CEST 2011 WEB-INF/classes/
     0 Thu Jun 16 18:50:48 CEST 2011 WEB-INF/classes/org/
     0 Thu Jun 16 18:50:48 CEST 2011 WEB-INF/classes/org/jboss/
     0 Thu Jun 16 18:50:48 CEST 2011 WEB-INF/classes/org/jboss/test/
     0 Thu Jun 16 18:50:48 CEST 2011 WEB-INF/classes/org/jboss/test/ws/
     0 Thu Jun 16 18:50:48 CEST 2011 WEB-INF/classes/org/jboss/test/ws/jaxws/
     0 Thu Jun 16 18:50:48 CEST 2011 WEB-INF/classes/org/jboss/test/ws/jaxws/samples/
     0 Thu Jun 16 18:50:48 CEST 2011 WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/
     0 Thu Jun 16 18:50:48 CEST 2011
WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/
     0 Thu Jun 16 18:50:48 CEST 2011
WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/jaas/
   982 Thu Jun 16 18:50:48 CEST 2011
WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/jaas/POJOEndpointAuthorizationIntercep
412 Thu Jun 16 18:50:48 CEST 2011
WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/jaas/ServiceIface.class
  1398 Thu Jun 16 18:50:48 CEST 2011
WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/jaas/ServiceImpl.class
     0 Thu Jun 16 18:50:48 CEST 2011
WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/jaxws/
   701 Thu Jun 16 18:50:48 CEST 2011
WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/jaxws/GreetMe.class
  1065 Thu Jun 16 18:50:48 CEST 2011
WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/jaxws/GreetMeResponse.class
   705 Thu Jun 16 18:50:48 CEST 2011
WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/jaxws/SayHello.class
  1069 Thu Jun 16 18:50:48 CEST 2011
WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/jaxws/SayHelloResponse.class
   556 Thu Jun 16 18:50:44 CEST 2011 WEB-INF/jaxws-endpoint-config.xml
   241 Thu Jun 16 18:50:44 CEST 2011 WEB-INF/jboss-web.xml
     0 Thu Jun 16 18:50:44 CEST 2011 WEB-INF/wsdl/
  3183 Thu Jun 16 18:50:44 CEST 2011 WEB-INF/wsdl/SecurityService.wsdl
  1012 Thu Jun 16 18:50:44 CEST 2011 WEB-INF/wsdl/SecurityService_schema1.xsd
```

If you're deploying the endpoint archive on WildFly, remember to add a dependency to *org.apache.ws.security* and *org.apache.cxf* module (due to the `@InInterceptor` annotation) in the MANIFEST.MF file.

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.7.1
Created-By: 17.0-b16 (Sun Microsystems Inc.)
Dependencies: org.apache.ws.security,org.apache.cxf
```

Client

Here too you start by consuming the published WSDL contract using the *wsconsume* tool. Then you simply invoke the the endpoint as a standard JAX-WS one:

```
QName serviceName = new QName("http://www.jboss.org/jbossws/ws-extensions/wssecuritypolicy",
"SecurityService");
URL wsdlURL = new URL(serviceURL + "?wsdl");
Service service = Service.create(wsdlURL, serviceName);
ServiceIface proxy = (ServiceIface)service.getPort(ServiceIface.class);

((BindingProvider)proxy).getRequestContext().put(SecurityConstants.USERNAME, "kermit");
((BindingProvider)proxy).getRequestContext().put(SecurityConstants.CALLBACK_HANDLER,
      "org.jboss.test.ws.jaxws.samples.wsse.policy.jaas.UsernamePasswordCallback");

proxy.sayHello();
```

The `UsernamePasswordCallback` class is shown below and is responsible for setting the passwords on client side just before performing the invocations:

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.jaas;

import java.io.IOException;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import org.apache.ws.security.WSPasswordCallback;

public class UsernamePasswordCallback implements CallbackHandler
{
   public void handle(Callback[] callbacks) throws IOException, UnsupportedCallbackException
   {
      WSPasswordCallback pc = (WSPasswordCallback)callbacks[0];
      if ("kermit".equals(pc.getIdentifier()))
         pc.setPassword("thefrog");
   }
}
```

If everything has been done properly, you should expect to calls to `sayHello()` fail when done with user "snoopy" and pass with user "kermit" (and credential "thefrog"); moreover, you should get an authorization error when trying to call `greetMe()` with user "kermit", as that does not have the "snoopies" role.

Secure transport

Another quite common use case is using WS-Security Username Token Profile over a secure transport (HTTPS). A scenario like this is implemented similarly to what's described in the previous example, except for few differences explained below.

First of all, here is an excerpt of a wsdl wth a sample security policy for Username Token over HTTPS:

```
...
```

```
<binding name="SecurityServicePortBinding" type="tns:ServiceIface">
  <wsp:PolicyReference URI="#SecurityServiceBindingPolicy"/>
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
  <operation name="sayHello">
    <soap:operation soapAction=""/>
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
  </operation>
</binding>
<service name="SecurityService">
   <port name="SecurityServicePort" binding="tns:SecurityServicePortBinding">
      <soap:address location="https://localhost:8443/jaxws-samples-wsse-policy-username"/>
   </port>
</service>

<wsp:Policy wsu:Id="SecurityServiceBindingPolicy">
   <wsp:ExactlyOne>
      <wsp:All>
         <foo:unknownPolicy xmlns:foo="http://cxf.apache.org/not/a/policy"/>
      </wsp:All>
      <wsp:All>
         <wsaws:UsingAddressing xmlns:wsaws="http://www.w3.org/2006/05/addressing/wsdl"/>
         <sp:TransportBinding>
            <wsp:Policy>
               <sp:TransportToken>
                  <wsp:Policy>
                     <sp:HttpsToken RequireClientCertificate="false"/>
                  </wsp:Policy>
               </sp:TransportToken>
               <sp:Layout>
                  <wsp:Policy>
                     <sp:Lax/>
                  </wsp:Policy>
               </sp:Layout>
               <sp:IncludeTimestamp/>
               <sp:AlgorithmSuite>
                  <wsp:Policy>
                     <sp:Basic128/>
                  </wsp:Policy>
               </sp:AlgorithmSuite>
            </wsp:Policy>
         </sp:TransportBinding>
         <sp:Wss10>
            <wsp:Policy>
               <sp:MustSupportRefKeyIdentifier/>
            </wsp:Policy>
         </sp:Wss10>
         <sp:SignedSupportingTokens>
            <wsp:Policy>
               <sp:UsernameToken
sp:IncludeToken="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy/IncludeToken/AlwaysToRecipie
<wsp:Policy>
                  <sp:WssUsernameToken10/>
               </wsp:Policy>
```

```
               </sp:UsernameToken>
            </wsp:Policy>
         </sp:SignedSupportingTokens>
      </wsp:All>
   </wsp:ExactlyOne>
</wsp:Policy>
```

The endpoint then needs of course to be actually available on HTTPS only, so we have a *web.xml* setting the *transport-guarantee* such as below:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app
   version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
   <servlet>
      <servlet-name>TestService</servlet-name>

<servlet-class>org.jboss.test.ws.jaxws.samples.wsse.policy.basic.ServiceImpl</servlet-class>
   </servlet>
   <servlet-mapping>
      <servlet-name>TestService</servlet-name>
      <url-pattern>/*</url-pattern>
   </servlet-mapping>

   <security-constraint>
    <web-resource-collection>
      <web-resource-name>TestService</web-resource-name>
      <url-pattern>/*</url-pattern>
    </web-resource-collection>
    <user-data-constraint>
      <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
  </security-constraint>
</web-app>
```

Secure conversation

Apache CXF supports WS-SecureConversation specification, which is about improving performance by allowing client and server to negotiate initial security keys to be used for later communication encryption/signature. This is done by having two policies in the wsdl contract, an outer one setting the security requirements to actually communicate with the endpoint and a bootstrap one, related to the communication for establishing the secure conversation keys. The client will be automatically sending an initial message to the server for negotiating the keys, then the actual communication to the endpoint takes place. As a consequence, Apache CXF needs a way to specify which WS-Security configuration properties are intended for the bootstrap policy and which are intended for the actual service policy. To accomplish this, properties intended for the bootstrap policy are appended with `.sct`.

```
...
((BindingProvider)proxy).getRequestContext().put("ws-security.signature.username.sct", "alice");
((BindingProvider)proxy).getRequestContext().put("ws-security.encryption.username.sct", "bob");
...
```

```
@WebService(
   ...
)
@EndpointProperties(value = {
      @EndpointProperty(key = "ws-security.encryption.properties.sct", value =
"bob.properties"),
      @EndpointProperty(key = "ws-security.signature.properties.sct", value = "bob.properties"),
      ...
      }
)
public class ServiceImpl implements ServiceIface {
   ...
}
```

Trust and STS

Trust overview

WS-Trust is a Web service specification that defines extensions to WS-Security. It is a general framework for implementing security in a distributed system. The standard is based on a centralized Security Token Service, STS, which is capable of authenticating clients and issuing tokens containing various kinds of authentication and authorization data. The specification describes a protocol used for issuance, exchange, and validation of security tokens, however the following specifications play an important role in the WS-Trust architecture: WS-SecurityPolicy 1.2, SAML 2.0, Username Token Profile, X.509 Token Profile, SAML Token Profile, and Kerberos Token Profile.

The WS-Trust extensions address the needs of applications that span multiple domains and requires the sharing of security keys by providing a standards based trusted third party web service (STS) to broker trust relationships between a Web service requester and a Web service provider. This architecture also alleviates the pain of service updates that require credential changes by providing a common location for this information. The STS is the common access point from which both the requester and provider retrieves and verifies security tokens.

There are three main components of the WS-Trust specification.

- The Security Token Service (STS), a web service that issues, renews, and validates security tokens.
- The message formats for security token requests and responses.
- The mechanisms for key exchange

Security Token Service

The Security Token Service, STS, is the core of the WS-Trust specification. It is a standards based mechanism for authentication and authorization. The STS is an implementation of the WS-Trust specification's protocol for issuing, exchanging, and validating security tokens, based on token format, namespace, or trust boundaries. The STS is a web service that acts as a trusted third party to broker trust relationships between a Web service requester and a Web service provider. It is a common access point trusted by both requester and provider to provide interoperable security tokens. It removes the need for a direct relationship between the two. Because the STS is a standards based mechanism for authentication, it helps ensure interoperability across realms and between different platforms.

The STS's WSDL contract defines how other applications and processes interact with it. In particular the WSDL defines the WS-Trust and WS-Security policies that a requester must fulfill in order to successfully communicate with the STS's endpoints. A web service requester consumes the STS's WSDL and with the aid of an STSClient utility, generates a message request compliant with the stated security policies and submits it to the STS endpoint. The STS validates the request and returns an appropriate response.

Apache CXF support

Apache CXF is an open-source, fully featured Web services framework. The JBossWS open source project integrates the JBoss Web Services (JBossWS) stack with the Apache CXF project modules thus providing WS-Trust and other JAX-WS functionality in WildFly. This integration makes it easy to deploy CXF STS implementations, however WildFly can run any WS-Trust compliant STS. In addition the Apache CXF API provides a STSClient utility to facilitate web service requester communication with its STS.

Detailed information about the Apache CXF's WS-Trust implementation can be found here.

A Basic WS-Trust Scenario

Here is an example of a basic WS-Trust scenario. It is comprised of a Web service requester (ws-requester), a Web service provider (ws-provider), and a Security Token Service (STS). The ws-provider requires a SAML 2.0 token issued from a designed STS to be presented by the ws-requester using asymmetric binding. These communication requirements are declared in the ws-provider's WSDL. The STS requires ws-requester credentials be provided in a WSS UsernameToken format request using symmetric binding. The STS's response is provided containing a SAML 2.0 token. These communication requirements are declared in the STS's WSDL.

1. A ws-requester contacts the ws-provider and consumes its WSDL. Upon finding the security token issuer requirement, it creates and configures a STSClient with the information it requires to generate a proper request.
2. The STSClient contacts the STS and consumes its WSDL. The security policies are discovered. The STSClient creates and sends an authentication request, with appropriate credentials.
3. The STS verifies the credentials.
4. In response, the STS issues a security token that provides proof that the ws-requester has authenticated with the STS.
5. The STClient presents a message with the security token to the ws-provider.
6. The ws-provider verifies the token was issued by the STS, thus proving the ws-requester has successfully authenticated with the STS.
7. The ws-provider executes the requested service and returns the results to the the ws-requester.

Web service provider

This section examines the crucial elements in providing endpoint security in the web service provider described in the basic WS-Trust scenario. The components that will be discussed are.

- web service provider's WSDL
- web service provider's Interface and Implementation classes.
- ServerCallbackHandler class
- Crypto properties and keystore files
- MANIFEST.MF

Web service provider WSDL

The web service provider is a contract-first endpoint. All the WS-trust and security policies for it are declared in the WSDL, SecurityService.wsdl. For this scenario a ws-requester is required to present a SAML 2.0 token issued from a designed STS. The address of the STS is provided in the WSDL. An asymmetric binding policy is used to encrypt and sign the SOAP body of messages that pass back and forth between ws-requester and ws-provider. X.509 certificates are use for the asymmetric binding. The rules for sharing the public and private keys in the SOAP request and response messages are declared. A detailed explanation of the security settings are provided in the comments in the listing below.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions targetNamespace="http://www.jboss.org/jbossws/ws-extensions/wssecuritypolicy"
name="SecurityService"
        xmlns:tns="http://www.jboss.org/jbossws/ws-extensions/wssecuritypolicy"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
```

```
        xmlns="http://schemas.xmlsoap.org/wsdl/"
        xmlns:wsp="http://www.w3.org/ns/ws-policy"
        xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"

xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
        xmlns:wsaws="http://www.w3.org/2005/08/addressing"
        xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702"
        xmlns:t="http://docs.oasis-open.org/ws-sx/ws-trust/200512">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://www.jboss.org/jbossws/ws-extensions/wssecuritypolicy"
schemaLocation="SecurityService_schema1.xsd"/>
    </xsd:schema>
  </types>
  <message name="sayHello">
    <part name="parameters" element="tns:sayHello"/>
  </message>
  <message name="sayHelloResponse">
    <part name="parameters" element="tns:sayHelloResponse"/>
  </message>
  <portType name="ServiceIface">
    <operation name="sayHello">
      <input message="tns:sayHello"/>
      <output message="tns:sayHelloResponse"/>
    </operation>
  </portType>
  <!--
        The wsp:PolicyReference binds the security requirments on all the STS endpoints.
        The wsp:Policy wsu:Id="#AsymmetricSAML2Policy" element is defined later in this file.
  -->
  <binding name="SecurityServicePortBinding" type="tns:ServiceIface">
    <wsp:PolicyReference URI="#AsymmetricSAML2Policy" />
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
    <operation name="sayHello">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="literal"/>
        <wsp:PolicyReference URI="#Input_Policy" />
      </input>
      <output>
        <soap:body use="literal"/>
        <wsp:PolicyReference URI="#Output_Policy" />
      </output>
    </operation>
  </binding>
  <service name="SecurityService">
    <port name="SecurityServicePort" binding="tns:SecurityServicePortBinding">
      <soap:address
location="http://@jboss.bind.address@:8080/jaxws-samples-wsse-policy-trust/SecurityService"/>
    </port>
  </service>

  <wsp:Policy wsu:Id="AsymmetricSAML2Policy">
        <wsp:ExactlyOne>
            <wsp:All>
  <!--
        The wsam:Addressing element, indicates that the endpoints of this
        web service MUST conform to the WS-Addressing specification.  The
```

```
        attribute wsp:Optional="false" enforces this assertion.
  -->
                <wsam:Addressing wsp:Optional="false">
                    <wsp:Policy />
                </wsam:Addressing>
  <!--
        The sp:AsymmetricBinding element indicates that security is provided
        at the SOAP layer. A public/private key combinations is required to
        protect the message.  The initiator will use it’s private key to sign
        the message and the recipient’s public key is used to encrypt the message.
        The recipient of the message will use it’s private key to decrypt it and
        initiator’s public key to verify the signature.
  -->
                <sp:AsymmetricBinding>
                    <wsp:Policy>
  <!--
        The sp:InitiatorToken element specifies the elements required in
        generating the initiator request to the ws-provider's service.
  -->
                        <sp:InitiatorToken>
                            <wsp:Policy>
  <!--
        The sp:IssuedToken element asserts that a SAML 2.0 security token is
        expected from the STS using a public key type.  The

sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/AlwaysToRe
attribute instructs the runtime to include the initiator's public key
        with every message sent to the recipient.

        The sp:RequestSecurityTokenTemplate element directs that all of the
        children of this element will be copied directly into the body of the
        RequestSecurityToken (RST) message that is sent to the STS when the
        initiator asks the STS to issue a token.
  -->
                                <sp:IssuedToken

sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/AlwaysToRe
<sp:RequestSecurityTokenTemplate>

<t:TokenType>http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV2.0</t:TokenType
<t:KeyType>http://docs.oasis-open.org/ws-sx/ws-trust/200512/PublicKey</t:KeyType>
                                    </sp:RequestSecurityTokenTemplate>
                                    <wsp:Policy>
                                        <sp:RequireInternalReference />
                                    </wsp:Policy>
  <!--
        The sp:Issuer element defines the STS's address and endpoint information
        This information is used by the STSClient.
  -->
                                    <sp:Issuer>

<wsaws:Address>http://@jboss.bind.address@:8080/jaxws-samples-wsse-policy-trust-sts/SecurityTokenS
<wsaws:Metadata xmlns:wsdli="http://www.w3.org/2006/01/wsdl-instance"

wsdli:wsdlLocation="http://@jboss.bind.address@:8080/jaxws-samples-wsse-policy-trust-sts/SecurityT
<wsaw:ServiceName xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl"

xmlns:stsns="http://docs.oasis-open.org/ws-sx/ws-trust/200512/"
```

```
EndpointName="UT_Port">stsns:SecurityTokenService</wsaw:ServiceName>
                                        </wsaws:Metadata>
                                    </sp:Issuer>
                                </sp:IssuedToken>
                            </wsp:Policy>
                        </sp:InitiatorToken>
  <!--
        The sp:RecipientToken element asserts the type of public/private key-pair
        expected from the recipient.  The

sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/Never">
        attribute indicates that the initiator's public key will never be included
        in the reply messages.

        The sp:WssX509V3Token10 element indicates that an X509 Version 3 token
        should be used in the message.
  -->
                        <sp:RecipientToken>
                            <wsp:Policy>
                                <sp:X509Token

sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/Never">
                                    <wsp:Policy>
                                        <sp:WssX509V3Token10 />
                                        <sp:RequireIssuerSerialReference />
                                    </wsp:Policy>
                                </sp:X509Token>
                            </wsp:Policy>
                        </sp:RecipientToken>
<!--
     The sp:Layout element,  indicates the layout rules to apply when adding
     items to the security header.  The sp:Lax sub-element indicates items
     are added to the security header in any order that conforms to
     WSS: SOAP Message Security.
-->
                        <sp:Layout>
                            <wsp:Policy>
                                <sp:Lax />
                            </wsp:Policy>
                        </sp:Layout>
                        <sp:IncludeTimestamp />
                        <sp:OnlySignEntireHeadersAndBody />
 <!--
     The sp:AlgorithmSuite element, requires the Basic256 algorithm suite
     be used in performing cryptographic operations.
-->
                        <sp:AlgorithmSuite>
                            <wsp:Policy>
                                <sp:Basic256 />
                            </wsp:Policy>
                        </sp:AlgorithmSuite>
                    </wsp:Policy>
                </sp:AsymmetricBinding>
<!--
    The sp:Wss11 element declares WSS: SOAP Message Security 1.1 options
    to be supported by the STS.  These particular elements generally refer
    to how keys are referenced within the SOAP envelope.  These are normally
```

```
    handled by CXF.
-->
                <sp:Wss11>
                    <wsp:Policy>
                        <sp:MustSupportRefIssuerSerial />
                        <sp:MustSupportRefThumbprint />
                        <sp:MustSupportRefEncryptedKey />
                    </wsp:Policy>
                </sp:Wss11>
<!--
    The sp:Trust13 element declares controls for WS-Trust 1.3 options.
    They are policy assertions related to exchanges specifically with
    client and server challenges and entropy behaviors.  Again these are
    normally handled by CXF.
-->
                <sp:Trust13>
                    <wsp:Policy>
                        <sp:MustSupportIssuedTokens />
                        <sp:RequireClientEntropy />
                        <sp:RequireServerEntropy />
                    </wsp:Policy>
                </sp:Trust13>
            </wsp:All>
        </wsp:ExactlyOne>
    </wsp:Policy>

    <wsp:Policy wsu:Id="Input_Policy">
        <wsp:ExactlyOne>
            <wsp:All>
                <sp:EncryptedParts>
                    <sp:Body />
                </sp:EncryptedParts>
                <sp:SignedParts>
                    <sp:Body />
                    <sp:Header Name="To" Namespace="http://www.w3.org/2005/08/addressing" />
                    <sp:Header Name="From" Namespace="http://www.w3.org/2005/08/addressing" />
                    <sp:Header Name="FaultTo" Namespace="http://www.w3.org/2005/08/addressing"
/>
                    <sp:Header Name="ReplyTo" Namespace="http://www.w3.org/2005/08/addressing"
/>
                    <sp:Header Name="MessageID" Namespace="http://www.w3.org/2005/08/addressing"
/>
                    <sp:Header Name="RelatesTo" Namespace="http://www.w3.org/2005/08/addressing"
/>
                    <sp:Header Name="Action" Namespace="http://www.w3.org/2005/08/addressing" />
                </sp:SignedParts>
            </wsp:All>
        </wsp:ExactlyOne>
    </wsp:Policy>

    <wsp:Policy wsu:Id="Output_Policy">
        <wsp:ExactlyOne>
            <wsp:All>
                <sp:EncryptedParts>
                    <sp:Body />
                </sp:EncryptedParts>
                <sp:SignedParts>
                    <sp:Body />
```

```
                    <sp:Header Name="To" Namespace="http://www.w3.org/2005/08/addressing" />
                    <sp:Header Name="From" Namespace="http://www.w3.org/2005/08/addressing" />
                    <sp:Header Name="FaultTo" Namespace="http://www.w3.org/2005/08/addressing"
/>
                    <sp:Header Name="ReplyTo" Namespace="http://www.w3.org/2005/08/addressing"
/>
                    <sp:Header Name="MessageID" Namespace="http://www.w3.org/2005/08/addressing"
/>
                    <sp:Header Name="RelatesTo" Namespace="http://www.w3.org/2005/08/addressing"
/>
                    <sp:Header Name="Action" Namespace="http://www.w3.org/2005/08/addressing" />
                </sp:SignedParts>
            </wsp:All>
        </wsp:ExactlyOne>
    </wsp:Policy>
</definitions>
```

Web service provider Interface

The web service provider interface class, ServiceIface, is a simple straight forward web service definition.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.service;

import javax.jws.WebMethod;
import javax.jws.WebService;

@WebService
(
   targetNamespace = "http://www.jboss.org/jbossws/ws-extensions/wssecuritypolicy"
)
public interface ServiceIface
{
   @WebMethod
   String sayHello();
}
```

Web service provider Implementation

The web service provider implementation class, ServiceImpl, is a simple POJO. It uses the standard WebService annotation to define the service endpoint. In addition there are two Apache CXF annotations, EndpointProperties and EndpointProperty used for configuring the endpoint for the CXF runtime. These annotations come from the Apache WSS4J project, which provides a Java implementation of the primary WS-Security standards for Web Services. These annotations are programmatically adding properties to the endpoint. With plain Apache CXF, these properties are often set via the <jaxws:properties> element on the <jaxws:endpoint> element in the Spring config; these annotations allow the properties to be configured in the code.

WSS4J uses the Crypto interface to get keys and certificates for encryption/decryption and for signature creation/verification. As is asserted by the WSDL, X509 keys and certificates are required for this service. The WSS4J configuration information being provided by ServiceImpl is for Crypto's Merlin implementation. More information will be provided about this in the keystore section.

The first EndpointProperty statement in the listing is declaring the user's name to use for the message signature. It is used as the alias name in the keystore to get the user's cert and private key for signature. The next two EndpointProperty statements declares the Java properties file that contains the (Merlin) crypto configuration information. In this case both for signing and encrypting the messages. WSS4J reads this file and extra required information for message handling. The last EndpointProperty statement declares the ServerCallbackHandler implementation class. It is used to obtain the user's password for the certificates in the keystore file.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.service;

import javax.jws.WebService;

import org.apache.cxf.annotations.EndpointProperties;
import org.apache.cxf.annotations.EndpointProperty;

@WebService
(
   portName = "SecurityServicePort",
   serviceName = "SecurityService",
   wsdlLocation = "WEB-INF/wsdl/SecurityService.wsdl",
   targetNamespace = "http://www.jboss.org/jbossws/ws-extensions/wssecuritypolicy",
   endpointInterface = "org.jboss.test.ws.jaxws.samples.wsse.policy.trust.service.ServiceIface"
)
@EndpointProperties(value = {
      @EndpointProperty(key = "ws-security.signature.username", value = "myservicekey"),
      @EndpointProperty(key = "ws-security.signature.properties", value =
"serviceKeystore.properties"),
      @EndpointProperty(key = "ws-security.encryption.properties", value =
"serviceKeystore.properties"),
      @EndpointProperty(key = "ws-security.callback-handler", value =
"org.jboss.test.ws.jaxws.samples.wsse.policy.trust.service.ServerCallbackHandler")
})
public class ServiceImpl implements ServiceIface
{
   public String sayHello()
   {
      return "WS-Trust Hello World!";
   }
}
```

ServerCallbackHandler

ServerCallbackHandler is a callback handler for the WSS4J Crypto API. It is used to obtain the password for the private key in the keystore. This class enables CXF to retrieve the password of the user name to use for the message signature. A certificates' password is not discoverable. The creator of the certificate must record the password he assigns and provide it when requested through the CallbackHandler. In this scenario skpass is the password for user myservicekey.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.service;

import java.util.HashMap;
import java.util.Map;

import org.jboss.wsf.stack.cxf.extensions.security.PasswordCallbackHandler;

public class ServerCallbackHandler extends PasswordCallbackHandler
{

   public ServerCallbackHandler()
   {
      super(getInitMap());
   }

   private static Map<String, String> getInitMap()
   {
      Map<String, String> passwords = new HashMap<String, String>();
      passwords.put("myservicekey", "skpass");
      return passwords;
   }
}
```

Crypto properties and keystore files

WSS4J's Crypto implementation is loaded and configured via a Java properties file that contains Crypto configuration data. The file contains implementation-specific properties such as a keystore location, password, default alias and the like. This application is using the Merlin implementation. File serviceKeystore.properties contains this information.

File servicestore.jks, is a Java KeyStore (JKS) repository. It contains self signed certificates for myservicekey and mystskey. *Self signed certificates are not appropriate for production use.*

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=sspass
org.apache.ws.security.crypto.merlin.keystore.alias=myservicekey
org.apache.ws.security.crypto.merlin.keystore.file=servicestore.jks
```

MANIFEST.MF

When deployed on WildFly this application requires access to the JBossWs and CXF APIs provided in module org.jboss.ws.cxf.jbossws-cxf-client. The dependency statement directs the server to provide them at deployment.

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.8.2
Created-By: 1.7.0_25-b15 (Oracle Corporation)
Dependencies: org.jboss.ws.cxf.jbossws-cxf-client
```

Security Token Service (STS)

This section examines the crucial elements in providing the Security Token Service functionality described in the basic WS-Trust scenario. The components that will be discussed are.

- STS's WSDL
- STS's implementation class.
- STSCallbackHandler class
- Crypto properties and keystore files
- MANIFEST.MF
- Server configuration files

STS WSDL

The STS is a contract-first endpoint. All the WS-trust and security policies for it are declared in the WSDL, ws-trust-1.4-service.wsdl. A symmetric binding policy is used to encrypt and sign the SOAP body of messages that pass back and forth between ws-requester and the STS. The ws-requester is required to authenticate itself by providing WSS UsernameToken credentials. The rules for sharing the public and private keys in the SOAP request and response messages are declared. A detailed explanation of the security settings are provided in the comments in the listing below.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
        targetNamespace="http://docs.oasis-open.org/ws-sx/ws-trust/200512/"
        xmlns:tns="http://docs.oasis-open.org/ws-sx/ws-trust/200512/"
        xmlns:wstrust="http://docs.oasis-open.org/ws-sx/ws-trust/200512/"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns:wsap10="http://www.w3.org/2006/05/addressing/wsdl"

xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
        xmlns:wsp="http://www.w3.org/ns/ws-policy"
    xmlns:wst="http://docs.oasis-open.org/ws-sx/ws-trust/200512"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata">

  <wsdl:types>
    <xs:schema elementFormDefault="qualified"
targetNamespace='http://docs.oasis-open.org/ws-sx/ws-trust/200512'>

      <xs:element name='RequestSecurityToken' type='wst:AbstractRequestSecurityTokenType' />
```

```
      <xs:element name='RequestSecurityTokenResponse'
type='wst:AbstractRequestSecurityTokenType' />

      <xs:complexType name='AbstractRequestSecurityTokenType' >
        <xs:sequence>
          <xs:any namespace='##any' processContents='lax' minOccurs='0' maxOccurs='unbounded' />
        </xs:sequence>
        <xs:attribute name='Context' type='xs:anyURI' use='optional' />
        <xs:anyAttribute namespace='##other' processContents='lax' />
      </xs:complexType>
      <xs:element name='RequestSecurityTokenCollection'
type='wst:RequestSecurityTokenCollectionType' />
      <xs:complexType name='RequestSecurityTokenCollectionType' >
        <xs:sequence>
          <xs:element name='RequestSecurityToken' type='wst:AbstractRequestSecurityTokenType'
minOccurs='2' maxOccurs='unbounded'/>
        </xs:sequence>
      </xs:complexType>

      <xs:element name='RequestSecurityTokenResponseCollection'
type='wst:RequestSecurityTokenResponseCollectionType' />
      <xs:complexType name='RequestSecurityTokenResponseCollectionType' >
        <xs:sequence>
          <xs:element ref='wst:RequestSecurityTokenResponse' minOccurs='1' maxOccurs='unbounded'
/>
        </xs:sequence>
        <xs:anyAttribute namespace='##other' processContents='lax' />
      </xs:complexType>

    </xs:schema>
  </wsdl:types>

  <!-- WS-Trust defines the following GEDs -->
  <wsdl:message name="RequestSecurityTokenMsg">
    <wsdl:part name="request" element="wst:RequestSecurityToken" />
  </wsdl:message>
  <wsdl:message name="RequestSecurityTokenResponseMsg">
    <wsdl:part name="response"
            element="wst:RequestSecurityTokenResponse" />
  </wsdl:message>
  <wsdl:message name="RequestSecurityTokenCollectionMsg">
    <wsdl:part name="requestCollection"
            element="wst:RequestSecurityTokenCollection"/>
  </wsdl:message>
  <wsdl:message name="RequestSecurityTokenResponseCollectionMsg">
    <wsdl:part name="responseCollection"
            element="wst:RequestSecurityTokenResponseCollection"/>
  </wsdl:message>

  <!-- This portType an example of a Requestor (or other) endpoint that
         Accepts SOAP-based challenges from a Security Token Service -->
  <wsdl:portType name="WSSecurityRequestor">
    <wsdl:operation name="Challenge">
      <wsdl:input message="tns:RequestSecurityTokenResponseMsg"/>
      <wsdl:output message="tns:RequestSecurityTokenResponseMsg"/>
    </wsdl:operation>
  </wsdl:portType>
```

```
  <!-- This portType is an example of an STS supporting full protocol -->
<!--
    The wsdl:portType and data types are XML elements defined by the
    WS_Trust specification.  The wsdl:portType defines the endpoints
    supported in the STS implementation.  This WSDL defines all operations
    that an STS implementation can support.
-->
  <wsdl:portType name="STS">
    <wsdl:operation name="Cancel">
      <wsdl:input wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Cancel"
message="tns:RequestSecurityTokenMsg"/>
      <wsdl:output
wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/CancelFinal"
message="tns:RequestSecurityTokenResponseMsg"/>
    </wsdl:operation>
    <wsdl:operation name="Issue">
      <wsdl:input wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Issue"
message="tns:RequestSecurityTokenMsg"/>
      <wsdl:output
wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTRC/IssueFinal"
message="tns:RequestSecurityTokenResponseCollectionMsg"/>
    </wsdl:operation>
    <wsdl:operation name="Renew">
      <wsdl:input wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Renew"
message="tns:RequestSecurityTokenMsg"/>
      <wsdl:output
wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/RenewFinal"
message="tns:RequestSecurityTokenResponseMsg"/>
    </wsdl:operation>
    <wsdl:operation name="Validate">
      <wsdl:input wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Validate"
message="tns:RequestSecurityTokenMsg"/>
      <wsdl:output
wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/ValidateFinal"
message="tns:RequestSecurityTokenResponseMsg"/>
    </wsdl:operation>
    <wsdl:operation name="KeyExchangeToken">
      <wsdl:input wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/KET"
message="tns:RequestSecurityTokenMsg"/>
      <wsdl:output wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/KETFinal"
message="tns:RequestSecurityTokenResponseMsg"/>
    </wsdl:operation>
    <wsdl:operation name="RequestCollection">
      <wsdl:input message="tns:RequestSecurityTokenCollectionMsg"/>
      <wsdl:output message="tns:RequestSecurityTokenResponseCollectionMsg"/>
    </wsdl:operation>
  </wsdl:portType>

  <!-- This portType is an example of an endpoint that accepts
         Unsolicited RequestSecurityTokenResponse messages -->
  <wsdl:portType name="SecurityTokenResponseService">
    <wsdl:operation name="RequestSecurityTokenResponse">
      <wsdl:input message="tns:RequestSecurityTokenResponseMsg"/>
    </wsdl:operation>
  </wsdl:portType>

<!--
```

```
    The wsp:PolicyReference binds the security requirments on all the STS endpoints.
    The wsp:Policy wsu:Id="UT_policy" element is later in this file.
-->
  <wsdl:binding name="UT_Binding" type="wstrust:STS">
    <wsp:PolicyReference URI="#UT_policy" />
      <soap:binding style="document"
          transport="http://schemas.xmlsoap.org/soap/http" />
      <wsdl:operation name="Issue">
          <soap:operation
              soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Issue" />
          <wsdl:input>
              <wsp:PolicyReference
               URI="#Input_policy" />
              <soap:body use="literal" />
          </wsdl:input>
          <wsdl:output>
              <wsp:PolicyReference
               URI="#Output_policy" />
              <soap:body use="literal" />
          </wsdl:output>
      </wsdl:operation>
      <wsdl:operation name="Validate">
          <soap:operation
              soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Validate" />
          <wsdl:input>
              <wsp:PolicyReference
               URI="#Input_policy" />
              <soap:body use="literal" />
          </wsdl:input>
          <wsdl:output>
              <wsp:PolicyReference
               URI="#Output_policy" />
              <soap:body use="literal" />
          </wsdl:output>
      </wsdl:operation>
      <wsdl:operation name="Cancel">
          <soap:operation
              soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Cancel" />
          <wsdl:input>
              <soap:body use="literal" />
          </wsdl:input>
          <wsdl:output>
              <soap:body use="literal" />
          </wsdl:output>
      </wsdl:operation>
      <wsdl:operation name="Renew">
          <soap:operation
              soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Renew" />
          <wsdl:input>
              <soap:body use="literal" />
          </wsdl:input>
          <wsdl:output>
              <soap:body use="literal" />
          </wsdl:output>
      </wsdl:operation>
      <wsdl:operation name="KeyExchangeToken">
          <soap:operation
              soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/KeyExchangeToken"
```

```
/>
          <wsdl:input>
              <soap:body use="literal" />
          </wsdl:input>
          <wsdl:output>
              <soap:body use="literal" />
          </wsdl:output>
      </wsdl:operation>
      <wsdl:operation name="RequestCollection">
          <soap:operation

soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/RequestCollection" />
          <wsdl:input>
              <soap:body use="literal" />
          </wsdl:input>
          <wsdl:output>
              <soap:body use="literal" />
          </wsdl:output>
      </wsdl:operation>
  </wsdl:binding>

  <wsdl:service name="SecurityTokenService">
      <wsdl:port name="UT_Port" binding="tns:UT_Binding">
         <soap:address location="http://localhost:8080/SecurityTokenService/UT" />
      </wsdl:port>
  </wsdl:service>

  <wsp:Policy wsu:Id="UT_policy">
      <wsp:ExactlyOne>
         <wsp:All>
<!--
    The sp:UsingAddressing element, indicates that the endpoints of this
    web service conforms to the WS-Addressing specification.  More detail
    can be found here: [http://www.w3.org/TR/2006/CR-ws-addr-wsdl-20060529]
-->
            <wsap10:UsingAddressing/>
<!--
    The sp:SymmetricBinding element indicates that security is provided
    at the SOAP layer and any initiator must authenticate itself by providing
    WSS UsernameToken credentials.
-->
            <sp:SymmetricBinding
               xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
               <wsp:Policy>
<!--
    In a symmetric binding, the keys used for encrypting and signing in both
    directions are derived from a single key, the one specified by the
    sp:ProtectionToken element.  The sp:X509Token sub-element declares this
    key to be a X.509 certificate and the
    IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/Never"
    attribute adds the requirement that the token MUST NOT be included in
    any messages sent between the initiator and the recipient; rather, an
    external reference to the token should be used.  Lastly the WssX509V3Token10
    sub-element declares that the Username token presented by the initiator
    should be compliant with Web Services Security UsernameToken Profile
    1.0 specification. [
http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0.pdf ]
-->
```

```
                  <sp:ProtectionToken>
                     <wsp:Policy>
                        <sp:X509Token

sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/Never">
                           <wsp:Policy>
                              <sp:RequireDerivedKeys />
                              <sp:RequireThumbprintReference />
                              <sp:WssX509V3Token10 />
                           </wsp:Policy>
                        </sp:X509Token>
                     </wsp:Policy>
                  </sp:ProtectionToken>
<!--
    The sp:AlgorithmSuite element, requires the Basic256 algorithm suite
    be used in performing cryptographic operations.
-->
                  <sp:AlgorithmSuite>
                     <wsp:Policy>
                        <sp:Basic256 />
                     </wsp:Policy>
                  </sp:AlgorithmSuite>
<!--
    The sp:Layout element,  indicates the layout rules to apply when adding
    items to the security header.  The sp:Lax sub-element indicates items
    are added to the security header in any order that conforms to
    WSS: SOAP Message Security.
-->
                  <sp:Layout>
                     <wsp:Policy>
                        <sp:Lax />
                     </wsp:Policy>
                  </sp:Layout>
                  <sp:IncludeTimestamp />
                  <sp:EncryptSignature />
                  <sp:OnlySignEntireHeadersAndBody />
               </wsp:Policy>
            </sp:SymmetricBinding>
<!--
    The sp:SignedSupportingTokens element declares that the security header
    of messages must contain a sp:UsernameToken and the token must be signed.
    The attribute
IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/AlwaysToRecip
on sp:UsernameToken indicates that the token MUST be included in all
    messages sent from initiator to the recipient and that the token MUST
    NOT be included in messages sent from the recipient to the initiator.
    And finally the element sp:WssUsernameToken10 is a policy assertion
    indicating the Username token should be as defined in  Web Services
    Security UsernameToken Profile 1.0
-->
            <sp:SignedSupportingTokens
               xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
               <wsp:Policy>
                  <sp:UsernameToken

sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/AlwaysToRe
<wsp:Policy>
                        <sp:WssUsernameToken10 />
```

```
                  </wsp:Policy>
               </sp:UsernameToken>
            </wsp:Policy>
         </sp:SignedSupportingTokens>
<!--
    The sp:Wss11 element declares WSS: SOAP Message Security 1.1 options
    to be supported by the STS.  These particular elements generally refer
    to how keys are referenced within the SOAP envelope.  These are normally
    handled by CXF.
-->
         <sp:Wss11
            xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
            <wsp:Policy>
               <sp:MustSupportRefKeyIdentifier />
               <sp:MustSupportRefIssuerSerial />
               <sp:MustSupportRefThumbprint />
               <sp:MustSupportRefEncryptedKey />
            </wsp:Policy>
         </sp:Wss11>
<!--
    The sp:Trust13 element declares controls for WS-Trust 1.3 options.
    They are policy assertions related to exchanges specifically with
    client and server challenges and entropy behaviors.  Again these are
    normally handled by CXF.
-->
         <sp:Trust13
            xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
            <wsp:Policy>
               <sp:MustSupportIssuedTokens />
               <sp:RequireClientEntropy />
               <sp:RequireServerEntropy />
            </wsp:Policy>
         </sp:Trust13>
      </wsp:All>
   </wsp:ExactlyOne>
</wsp:Policy>

<wsp:Policy wsu:Id="Input_policy">
   <wsp:ExactlyOne>
      <wsp:All>
         <sp:SignedParts
            xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
            <sp:Body />
            <sp:Header Name="To"
               Namespace="http://www.w3.org/2005/08/addressing" />
            <sp:Header Name="From"
               Namespace="http://www.w3.org/2005/08/addressing" />
            <sp:Header Name="FaultTo"
               Namespace="http://www.w3.org/2005/08/addressing" />
            <sp:Header Name="ReplyTo"
               Namespace="http://www.w3.org/2005/08/addressing" />
            <sp:Header Name="MessageID"
               Namespace="http://www.w3.org/2005/08/addressing" />
            <sp:Header Name="RelatesTo"
               Namespace="http://www.w3.org/2005/08/addressing" />
            <sp:Header Name="Action"
               Namespace="http://www.w3.org/2005/08/addressing" />
         </sp:SignedParts>
```

```
            <sp:EncryptedParts
               xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
               <sp:Body />
            </sp:EncryptedParts>
         </wsp:All>
      </wsp:ExactlyOne>
   </wsp:Policy>

   <wsp:Policy wsu:Id="Output_policy">
      <wsp:ExactlyOne>
         <wsp:All>
            <sp:SignedParts
               xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
               <sp:Body />
               <sp:Header Name="To"
                  Namespace="http://www.w3.org/2005/08/addressing" />
               <sp:Header Name="From"
                  Namespace="http://www.w3.org/2005/08/addressing" />
               <sp:Header Name="FaultTo"
                  Namespace="http://www.w3.org/2005/08/addressing" />
               <sp:Header Name="ReplyTo"
                  Namespace="http://www.w3.org/2005/08/addressing" />
               <sp:Header Name="MessageID"
                  Namespace="http://www.w3.org/2005/08/addressing" />
               <sp:Header Name="RelatesTo"
                  Namespace="http://www.w3.org/2005/08/addressing" />
               <sp:Header Name="Action"
                  Namespace="http://www.w3.org/2005/08/addressing" />
            </sp:SignedParts>
            <sp:EncryptedParts
               xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
               <sp:Body />
            </sp:EncryptedParts>
         </wsp:All>
      </wsp:ExactlyOne>
   </wsp:Policy>

</wsdl:definitions>
```

STS Implementation

The Apache CXF's STS, SecurityTokenServiceProvider, is a web service provider that is compliant with the protocols and functionality defined by the WS-Trust specification. It has a modular architecture. Many of its components are configurable or replaceable and there are many optional features that are enabled by implementing and configuring plug-ins. Users can customize their own STS by extending from SecurityTokenServiceProvider and overriding the default settings. Extensive information about the CXF's STS configurable and pluggable components can be found here.

This STS implementation class, SimpleSTS, is a POJO that extends from SecurityTokenServiceProvider. Note that the class is defined with a WebServiceProvider annotation and not a WebService annotation. This annotation defines the service as a Provider-based endpoint, meaning it supports a more messaging-oriented approach to Web services. In particular, it signals that the exchanged messages will be XML documents of some type. SecurityTokenServiceProvider is an implementation of the javax.xml.ws.Provider interface. In comparison the WebService annotation defines a (service endpoint interface) SEI-based endpoint which supports message exchange via SOAP envelopes.

As was done in the ServiceImpl class, the WSS4J annotations EndpointProperties and EndpointProperty are providing endpoint configuration for the CXF runtime. This was previous described here.

The InInterceptors annotation is used to specify a JBossWS integration interceptor to be used for authenticating incoming requests; JAAS integration is used here for authentication, the username/passoword coming from the UsernameToken in the ws-requester message are used for authenticating the requester against a security domain on the application server hosting the STS deployment.

In this implementation we are customizing the operations of token issuance, token validation and their static properties.

StaticSTSProperties is used to set select properties for configuring resources in the STS. You may think this is a duplication of the settings made with the WSS4J annotations. The values are the same but the underlaying structures being set are different, thus this information must be declared in both places.

The setIssuer setting is important because it uniquely identifies the issuing STS. The issuer string is embedded in issued tokens and, when validating tokens, the STS checks the issuer string value. Consequently, it is important to use the issuer string in a consistent way, so that the STS can recognize the tokens that it has issued.

The setEndpoints call allows the declaration of a set of allowed token recipients by address. The addresses are specified as reg-ex patterns.

TokenIssueOperation and TokenValidateOperation have a modular structure. This allows custom behaviors to be injected into the processing of messages. In this case we are overriding the SecurityTokenServiceProvider's default behavior and performing SAML token processing and validation. CXF provides an implementation of a SAMLTokenProvider and SAMLTokenValidator which we are using rather than writing our own.

Learn more about the SAMLTokenProvider here.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust;

import java.util.Arrays;
import java.util.LinkedList;
import java.util.List;

import javax.xml.ws.WebServiceProvider;

import org.apache.cxf.annotations.EndpointProperties;
import org.apache.cxf.annotations.EndpointProperty;
import org.apache.cxf.interceptor.InInterceptors;
import org.apache.cxf.sts.StaticSTSProperties;
```

```
import org.apache.cxf.sts.operation.TokenIssueOperation;
import org.apache.cxf.sts.operation.TokenValidateOperation;
import org.apache.cxf.sts.service.ServiceMBean;
import org.apache.cxf.sts.service.StaticService;
import org.apache.cxf.sts.token.provider.SAMLTokenProvider;
import org.apache.cxf.sts.token.validator.SAMLTokenValidator;
import org.apache.cxf.ws.security.sts.provider.SecurityTokenServiceProvider;

@WebServiceProvider(serviceName = "SecurityTokenService",
      portName = "UT_Port",
      targetNamespace = "http://docs.oasis-open.org/ws-sx/ws-trust/200512/",
      wsdlLocation = "WEB-INF/wsdl/ws-trust-1.4-service.wsdl")
@EndpointProperties(value = {
      @EndpointProperty(key = "ws-security.signature.username", value = "mystskey"),
      @EndpointProperty(key = "ws-security.signature.properties", value =
"stsKeystore.properties"),
      @EndpointProperty(key = "ws-security.callback-handler", value =
"org.jboss.test.ws.jaxws.samples.wsse.policy.trust.STSCallbackHandler"),
      //to let the JAAS integration deal with validation through the interceptor below
      @EndpointProperty(key = "ws-security.validate.token", value = "false")

})
@InInterceptors(interceptors =
{"org.jboss.wsf.stack.cxf.security.authentication.SubjectCreatingPolicyInterceptor"})
public class SampleSTS extends SecurityTokenServiceProvider
{
   public SampleSTS() throws Exception
   {
      super();

      StaticSTSProperties props = new StaticSTSProperties();
      props.setSignaturePropertiesFile("stsKeystore.properties");
      props.setSignatureUsername("mystskey");
      props.setCallbackHandlerClass(STSCallbackHandler.class.getName());
      props.setIssuer("DoubleItSTSIssuer");

      List<ServiceMBean> services = new LinkedList<ServiceMBean>();
      StaticService service = new StaticService();
      service.setEndpoints(Arrays.asList(
              "http://localhost:(\\d)*/jaxws-samples-wsse-policy-trust/SecurityService",
              "http://\\[::1\\]:(\\d)*/jaxws-samples-wsse-policy-trust/SecurityService",

"http://\\[0:0:0:0:0:0:0:1\\]:(\\d)*/jaxws-samples-wsse-policy-trust/SecurityService"
              ));
      services.add(service);

      TokenIssueOperation issueOperation = new TokenIssueOperation();
      issueOperation.setServices(services);
      issueOperation.getTokenProviders().add(new SAMLTokenProvider());
      issueOperation.setStsProperties(props);

      TokenValidateOperation validateOperation = new TokenValidateOperation();
      validateOperation.getTokenValidators().add(new SAMLTokenValidator());
      validateOperation.setStsProperties(props);

      this.setIssueOperation(issueOperation);
      this.setValidateOperation(validateOperation);
   }
```

```
}
```

STSCallbackHandler

STSCallbackHandler is a callback handler for the WSS4J Crypto API. It is used to obtain the password for the private key in the keystore. This class enables CXF to retrieve the password of the user name to use for the message signature.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.sts;

import java.util.HashMap;
import java.util.Map;

import org.jboss.wsf.stack.cxf.extensions.security.PasswordCallbackHandler;

public class STSCallbackHandler extends PasswordCallbackHandler
{
   public STSCallbackHandler()
   {
      super(getInitMap());
   }

   private static Map<String, String> getInitMap()
   {
      Map<String, String> passwords = new HashMap<String, String>();
      passwords.put("mystskey", "stskpass");
      return passwords;
   }
}
```

Crypto properties and keystore files

WSS4J's Crypto implementation is loaded and configured via a Java properties file that contains Crypto configuration data. The file contains implementation-specific properties such as a keystore location, password, default alias and the like. This application is using the Merlin implementation. File stsKeystore.properties contains this information.

File servicestore.jks, is a Java KeyStore (JKS) repository. It contains self signed certificates for myservicekey and mystskey. *Self signed certificates are not appropriate for production use.*

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=stsspass
org.apache.ws.security.crypto.merlin.keystore.file=stsstore.jks
```

MANIFEST.MF

When deployed on WildFly, this application requires access to the JBossWs and CXF APIs provided in modules org.jboss.ws.cxf.jbossws-cxf-client and org.apache.cxf. The Apache CXF internals, org.apache.cxf.impl, are needed to build the STS configuration in the `SampleSTS` constructor. The dependency statement directs the server to provide them at deployment.

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.8.2
Created-By: 1.7.0_25-b15 (Oracle Corporation)
Dependencies: org.jboss.ws.cxf.jbossws-cxf-client,org.apache.cxf.impl
```

Security Domain

The STS requires a JBoss security domain be configured. The jboss-web.xml descriptor declares a named security domain,"JBossWS-trust-sts" to be used by this service for authentication. This security domain requires two properties files and the addition of a security-domain declaration in the JBoss server configuration file.

For this scenario the domain needs to contain user *alice*, password *clarinet*, and role *friend*. See the listings below for jbossws-users.properties and jbossws-roles.properties. In addition the following XML must be added to the JBoss security subsystem in the server configuration file. Replace "**SOME_PATH**" with appropriate information.

```
<security-domain name="JBossWS-trust-sts">
  <authentication>
    <login-module code="UsersRoles" flag="required">
      <module-option name="usersProperties" value="/SOME_PATH/jbossws-users.properties"/>
      <module-option name="unauthenticatedIdentity" value="anonymous"/>
      <module-option name="rolesProperties" value="/SOME_PATH/jbossws-roles.properties"/>
    </login-module>
  </authentication>
</security-domain>
```

jboss-web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jboss-web PUBLIC "-//JBoss//DTD Web Application 2.4//EN" ">
<jboss-web>
  <security-domain>java:/jaas/JBossWS-trust-sts</security-domain>
</jboss-web>
```

jbossws-users.properties

```
# A sample users.properties file for use with the UsersRolesLoginModule
alice=clarinet
```

jbossws-roles.properties

```
# A sample roles.properties file for use with the UsersRolesLoginModule
alice=friend
```

WS-MetadataExchange and interoperability

To achieve better interoperability, you might consider allowing the STS endpoint to reply to WS-MetadataExchange messages directed to the `/mex` URL sub-path (e.g. http://localhost:8080/jaxws-samples-wsse-policy-trust-sts/SecurityTokenService/mex). This can be done by tweaking the *url-pattern* for the underlying endpoint servlet, for instance by adding a *web.xml* descriptor as follows to the deployment:<?xml version="1.0" encoding="UTF-8"?>
<web-app
version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
<servlet>
<servlet-name>TestSecurityTokenService</servlet-name>
<servlet-class>org.jboss.test.ws.jaxws.samples.wsse.policy.trust.SampleSTS</servlet-class>
</servlet>
<servlet-mapping>
<servlet-name>TestSecurityTokenService</servlet-name>
<url-pattern>/SecurityTokenService/*</url-pattern>
</servlet-mapping>
</web-app>
As a matter of fact, at the time of writing some webservices implementations (including *Metro*) assume the `/mex` URL as the default choice for directing WS-MetadataExchange requests to and use that to retrieve STS wsdl contracts.

Web service requester

This section examines the crucial elements in calling a web service that implements endpoint security as described in the basic WS-Trust scenario. The components that will be discussed are.

- web service requester's implementation
- ClientCallbackHandler
- Crypto properties and keystore files

Web service requester Implementation
The ws-requester, the client, uses standard procedures for creating a reference to the web service in the first four line. To address the endpoint security requirements, the web service's "Request Context" is configured with the information needed in message generation. In addition, the STSClient that communicates with the STS is configured with similar values. Note the key strings ending with a ".it" suffix. This suffix flags these settings as belonging to the STSClient. The internal CXF code assigns this information to the STSClient that is auto-generated for this service call.

There is an alternate method of setting up the STSCLient. The user may provide their own instance of the STSClient. The CXF code will use this object and not auto-generate one. This is used in the ActAs and OnBehalfOf examples. When providing the STSClient in this way, the user must provide a org.apache.cxf.Bus for it and the configuration keys must not have the ".it" suffix.

```
QName serviceName = new QName("http://www.jboss.org/jbossws/ws-extensions/wssecuritypolicy",
"SecurityService");
URL wsdlURL = new URL(serviceURL + "?wsdl");
Service service = Service.create(wsdlURL, serviceName);
ServiceIface proxy = (ServiceIface) service.getPort(ServiceIface.class);

// set the security related configuration information for the service "request"
Map<String, Object> ctx = ((BindingProvider) proxy).getRequestContext();
ctx.put(SecurityConstants.CALLBACK_HANDLER, new ClientCallbackHandler());
ctx.put(SecurityConstants.SIGNATURE_PROPERTIES,
   Thread.currentThread().getContextClassLoader().getResource(
   "META-INF/clientKeystore.properties"));
ctx.put(SecurityConstants.ENCRYPT_PROPERTIES,
   Thread.currentThread().getContextClassLoader().getResource(
   "META-INF/clientKeystore.properties"));
ctx.put(SecurityConstants.SIGNATURE_USERNAME, "myclientkey");
ctx.put(SecurityConstants.ENCRYPT_USERNAME, "myservicekey");


//-- Configuration settings that will be transfered to the STSClient
// "alice" is the name provided for the WSS Username. Her password will
// be retreived from the ClientCallbackHander by the STSClient.
ctx.put(SecurityConstants.USERNAME + ".it", "alice");
ctx.put(SecurityConstants.CALLBACK_HANDLER + ".it", new ClientCallbackHandler());
ctx.put(SecurityConstants.ENCRYPT_PROPERTIES + ".it",
   Thread.currentThread().getContextClassLoader().getResource(
   "META-INF/clientKeystore.properties"));
ctx.put(SecurityConstants.ENCRYPT_USERNAME + ".it", "mystskey");
// alias name in the keystore to get the user's public key to send to the STS
ctx.put(SecurityConstants.STS_TOKEN_USERNAME + ".it", "myclientkey");
// Crypto property configuration to use for the STS
ctx.put(SecurityConstants.STS_TOKEN_PROPERTIES + ".it",
   Thread.currentThread().getContextClassLoader().getResource(
   "META-INF/clientKeystore.properties"));
// write out an X509Certificate structure in UseKey/KeyInfo
ctx.put(SecurityConstants.STS_TOKEN_USE_CERT_FOR_KEYINFO + ".it", "true");
// Setting indicates the  STSclient should not try using the WS-MetadataExchange
// call using STS EPR WSA address when the endpoint contract does not contain
// WS-MetadataExchange info.
ctx.put("ws-security.sts.disable-wsmex-call-using-epr-address", "true");

proxy.sayHello();
```

ClientCallbackHandler

ClientCallbackHandler is a callback handler for the WSS4J Crypto API. It is used to obtain the password for the private key in the keystore. This class enables CXF to retrieve the password of the user name to use for the message signature. Note that "alice" and her password have been provided here. This information is not in the (JKS) keystore but provided in the WildFly security domain. It was declared in file jbossws-users.properties.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.shared;

import java.io.IOException;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import org.apache.ws.security.WSPasswordCallback;

public class ClientCallbackHandler implements CallbackHandler {

    public void handle(Callback[] callbacks) throws IOException,
            UnsupportedCallbackException {
        for (int i = 0; i < callbacks.length; i++) {
            if (callbacks[i] instanceof WSPasswordCallback) {
                WSPasswordCallback pc = (WSPasswordCallback) callbacks[i];
                if ("myclientkey".equals(pc.getIdentifier())) {
                    pc.setPassword("ckpass");
                    break;
                } else if ("alice".equals(pc.getIdentifier())) {
                    pc.setPassword("clarinet");
                    break;
                }
            }
        }
    }
}
```

Requester Crypto properties and keystore files

WSS4J's Crypto implementation is loaded and configured via a Java properties file that contains Crypto configuration data. The file contains implementation-specific properties such as a keystore location, password, default alias and the like. This application is using the Merlin implementation. File clientKeystore.properties contains this information.

File clientstore.jks, is a Java KeyStore (JKS) repository. It contains self signed certificates for myservicekey and mystskey. *Self signed certificates are not appropriate for production use.*

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=cspass
org.apache.ws.security.crypto.merlin.keystore.alias=myclientkey
org.apache.ws.security.crypto.merlin.keystore.file=META-INF/clientstore.jks
```

PicketLink STS

PicketLink provides facilities for building up an alternative to the Apache CXF Security Token Service implementation.

Similarly to the previous implementation, the STS is served through a WebServiceProvider annotated POJO:

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust;

import javax.annotation.Resource;
import javax.xml.ws.Service;
import javax.xml.ws.ServiceMode;
import javax.xml.ws.WebServiceContext;
import javax.xml.ws.WebServiceProvider;

import org.apache.cxf.annotations.EndpointProperties;
import org.apache.cxf.annotations.EndpointProperty;
import org.apache.cxf.interceptor.InInterceptors;
import org.picketlink.identity.federation.core.wstrust.PicketLinkSTS;

@WebServiceProvider(serviceName = "PicketLinkSTS", portName = "PicketLinkSTSPort",
targetNamespace = "urn:picketlink:identity-federation:sts", wsdlLocation =
"WEB-INF/wsdl/PicketLinkSTS.wsdl")
@ServiceMode(value = Service.Mode.MESSAGE)
//be sure to have dependency on org.apache.cxf module when on AS7, otherwise Apache CXF
annotations are ignored
@EndpointProperties(value = {
@EndpointProperty(key = "ws-security.signature.username", value = "mystskey"),
@EndpointProperty(key = "ws-security.signature.properties", value = "stsKeystore.properties"),
@EndpointProperty(key = "ws-security.callback-handler", value =
"org.jboss.test.ws.jaxws.samples.wsse.policy.trust.STSCallbackHandler"),
@EndpointProperty(key = "ws-security.validate.token", value = "false") //to let the JAAS
integration deal with validation through the interceptor below
})
@InInterceptors(interceptors =

)
public class PicketLinkSTService extends PicketLinkSTS {
@Resource
public void setWSC(WebServiceContext wctx)
Unknown macro: { this.context = wctx; }

}
```

The `@WebServiceProvider` annotation references the following WS-Policy enabled wsdl contract; please note the wsdl operations, messages and such must match the `PicketLinkSTS` implementation:

```
<?xml version="1.0"?>
<wsdl:definitions name="PicketLinkSTS" targetNamespace="urn:picketlink:identity-federation:sts"
    xmlns:tns="urn:picketlink:identity-federation:sts"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsap10="http://www.w3.org/2006/05/addressing/wsdl"
    xmlns:wsp="http://www.w3.org/ns/ws-policy"

xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
```

```
    xmlns:wst="http://docs.oasis-open.org/ws-sx/ws-trust/200512"
    xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/">
  <wsdl:types>
    <xs:schema elementFormDefault="qualified"
targetNamespace='http://docs.oasis-open.org/ws-sx/ws-trust/200512'
xmlns:xs="http://www.w3.org/2001/XMLSchema">
      <xs:element name='RequestSecurityToken' type='wst:AbstractRequestSecurityTokenType' />
      <xs:element name='RequestSecurityTokenResponse'
type='wst:AbstractRequestSecurityTokenType' />
      <xs:complexType name='AbstractRequestSecurityTokenType' >
        <xs:sequence>
          <xs:any namespace='##any' processContents='lax' minOccurs='0' maxOccurs='unbounded' />
        </xs:sequence>
        <xs:attribute name='Context' type='xs:anyURI' use='optional' />
        <xs:anyAttribute namespace='##other' processContents='lax' />
      </xs:complexType>
      <xs:element name='RequestSecurityTokenCollection'
type='wst:RequestSecurityTokenCollectionType' />
      <xs:complexType name='RequestSecurityTokenCollectionType' >
        <xs:sequence>
          <xs:element name='RequestSecurityToken' type='wst:AbstractRequestSecurityTokenType'
minOccurs='2' maxOccurs='unbounded'/>
        </xs:sequence>
      </xs:complexType>
      <xs:element name='RequestSecurityTokenResponseCollection'
type='wst:RequestSecurityTokenResponseCollectionType' />
      <xs:complexType name='RequestSecurityTokenResponseCollectionType' >
        <xs:sequence>
          <xs:element ref='wst:RequestSecurityTokenResponse' minOccurs='1' maxOccurs='unbounded'
/>
        </xs:sequence>
        <xs:anyAttribute namespace='##other' processContents='lax' />
      </xs:complexType>
    </xs:schema>
  </wsdl:types>

  <wsdl:message name="RequestSecurityTokenMsg">
    <wsdl:part name="request" element="wst:RequestSecurityToken" />
  </wsdl:message>
  <wsdl:message name="RequestSecurityTokenResponseCollectionMsg">
    <wsdl:part name="responseCollection"
            element="wst:RequestSecurityTokenResponseCollection"/>
  </wsdl:message>

  <wsdl:portType name="SecureTokenService">
    <wsdl:operation name="IssueToken">
      <wsdl:input wsap10:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Issue"
message="tns:RequestSecurityTokenMsg"/>
      <wsdl:output
wsap10:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTRC/IssueFinal"
message="tns:RequestSecurityTokenResponseCollectionMsg"/>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="STSBinding" type="tns:SecureTokenService">
    <wsp:PolicyReference URI="#UT_policy" />
    <soap12:binding transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="IssueToken">
      <soap12:operation soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Issue"
```

```
style="document"/>
      <wsdl:input>
        <wsp:PolicyReference URI="#Input_policy" />
        <soap12:body use="literal"/>
      </wsdl:input>
      <wsdl:output>
        <wsp:PolicyReference URI="#Output_policy" />
        <soap12:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="PicketLinkSTS">
    <wsdl:port name="PicketLinkSTSPort" binding="tns:STSBinding">
      <soap12:address location="http://localhost:8080/picketlink-sts/PicketLinkSTS"/>
    </wsdl:port>
  </wsdl:service>

  <wsp:Policy wsu:Id="UT_policy">
      <wsp:ExactlyOne>
         <wsp:All>
            <wsap10:UsingAddressing/>
            <sp:SymmetricBinding
               xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
               <wsp:Policy>
                  <sp:ProtectionToken>
                     <wsp:Policy>
                        <sp:X509Token

sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/Never">
                           <wsp:Policy>
                              <sp:RequireDerivedKeys />
                              <sp:RequireThumbprintReference />
                              <sp:WssX509V3Token10 />
                           </wsp:Policy>
                        </sp:X509Token>
                     </wsp:Policy>
                  </sp:ProtectionToken>
                  <sp:AlgorithmSuite>
                     <wsp:Policy>
                        <sp:Basic256 />
                     </wsp:Policy>
                  </sp:AlgorithmSuite>
                  <sp:Layout>
                     <wsp:Policy>
                        <sp:Lax />
                     </wsp:Policy>
                  </sp:Layout>
                  <sp:IncludeTimestamp />
                  <sp:EncryptSignature />
                  <sp:OnlySignEntireHeadersAndBody />
               </wsp:Policy>
            </sp:SymmetricBinding>
            <sp:SignedSupportingTokens
               xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
               <wsp:Policy>
                  <sp:UsernameToken

sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/AlwaysToRe
```

```
<wsp:Policy>
                        <sp:WssUsernameToken10 />
                     </wsp:Policy>
                  </sp:UsernameToken>
               </wsp:Policy>
            </sp:SignedSupportingTokens>
            <sp:Wss11
               xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
               <wsp:Policy>
                  <sp:MustSupportRefKeyIdentifier />
                  <sp:MustSupportRefIssuerSerial />
                  <sp:MustSupportRefThumbprint />
                  <sp:MustSupportRefEncryptedKey />
               </wsp:Policy>
            </sp:Wss11>
            <sp:Trust13
               xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
               <wsp:Policy>
                  <sp:MustSupportIssuedTokens />
                  <sp:RequireClientEntropy />
                  <sp:RequireServerEntropy />
               </wsp:Policy>
            </sp:Trust13>
         </wsp:All>
      </wsp:ExactlyOne>
   </wsp:Policy>

   <wsp:Policy wsu:Id="Input_policy">
      <wsp:ExactlyOne>
         <wsp:All>
            <sp:SignedParts
               xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
               <sp:Body />
               <sp:Header Name="To"
                  Namespace="http://www.w3.org/2005/08/addressing" />
               <sp:Header Name="From"
                  Namespace="http://www.w3.org/2005/08/addressing" />
               <sp:Header Name="FaultTo"
                  Namespace="http://www.w3.org/2005/08/addressing" />
               <sp:Header Name="ReplyTo"
                  Namespace="http://www.w3.org/2005/08/addressing" />
               <sp:Header Name="MessageID"
                  Namespace="http://www.w3.org/2005/08/addressing" />
               <sp:Header Name="RelatesTo"
                  Namespace="http://www.w3.org/2005/08/addressing" />
               <sp:Header Name="Action"
                  Namespace="http://www.w3.org/2005/08/addressing" />
            </sp:SignedParts>
            <sp:EncryptedParts
               xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
               <sp:Body />
            </sp:EncryptedParts>
         </wsp:All>
      </wsp:ExactlyOne>
   </wsp:Policy>

   <wsp:Policy wsu:Id="Output_policy">
      <wsp:ExactlyOne>
```

```
         <wsp:All>
            <sp:SignedParts
               xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
               <sp:Body />
               <sp:Header Name="To"
                  Namespace="http://www.w3.org/2005/08/addressing" />
               <sp:Header Name="From"
                  Namespace="http://www.w3.org/2005/08/addressing" />
               <sp:Header Name="FaultTo"
                  Namespace="http://www.w3.org/2005/08/addressing" />
               <sp:Header Name="ReplyTo"
                  Namespace="http://www.w3.org/2005/08/addressing" />
               <sp:Header Name="MessageID"
                  Namespace="http://www.w3.org/2005/08/addressing" />
               <sp:Header Name="RelatesTo"
                  Namespace="http://www.w3.org/2005/08/addressing" />
               <sp:Header Name="Action"
                  Namespace="http://www.w3.org/2005/08/addressing" />
            </sp:SignedParts>
            <sp:EncryptedParts
               xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
               <sp:Body />
            </sp:EncryptedParts>
         </wsp:All>
      </wsp:ExactlyOne>
   </wsp:Policy>

</wsdl:definitions>
```

Differently from the Apache CXF STS example described above, the PicketLink based STS gets its configuration from a picketlink-sts.xml descriptor which must be added in WEB-INF into the deployment; please refer to the PicketLink documentation for further information:

```
<PicketLinkSTS xmlns="urn:picketlink:identity-federation:config:1.0"
    STSName="PicketLinkSTS" TokenTimeout="7200" EncryptToken="false">
    <KeyProvider ClassName="org.picketlink.identity.federation.core.impl.KeyStoreKeyManager">
        <Auth Key="KeyStoreURL" Value="stsstore.jks"/>
        <Auth Key="KeyStorePass" Value="stsspass"/>
        <Auth Key="SigningKeyAlias" Value="mystskey"/>
        <Auth Key="SigningKeyPass" Value="stskpass"/>
        <ValidatingAlias
Key="http://localhost:8080/jaxws-samples-wsse-policy-trust/SecurityService"
Value="myservicekey"/>
    </KeyProvider>
    <TokenProviders>
            <TokenProvider
ProviderClass="org.picketlink.identity.federation.core.wstrust.plugins.saml.SAML11TokenProvider"

TokenType="http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV1.1"
            TokenElement="Assertion"
            TokenElementNS="urn:oasis:names:tc:SAML:1.0:assertion"/>
            <TokenProvider
ProviderClass="org.picketlink.identity.federation.core.wstrust.plugins.saml.SAML20TokenProvider"

TokenType="http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV2.0"
            TokenElement="Assertion"
            TokenElementNS="urn:oasis:names:tc:SAML:2.0:assertion"/>
    </TokenProviders>
</PicketLinkSTS>
```

Finally, the PicketLink alternative approach of course requires different WildFly module dependencies to be declared in the MANIFEST.MF:

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.8.2
Created-By: 1.6.0_26-b03 (Sun Microsystems Inc.)
Dependencies: org.apache.ws.security,org.apache.cxf,org.picketlink
```

Here is how the PicketLink STS endpoint is packaged:

```
alessio@inuyasha /dati/jbossws/stack/cxf/trunk $ jar -tvf
./modules/testsuite/cxf-tests/target/test-libs/jaxws-samples-wsse-policy-trustPicketLink-sts.war
     0 Mon Sep 03 17:38:38 CEST 2012 META-INF/
   174 Mon Sep 03 17:38:36 CEST 2012 META-INF/MANIFEST.MF
     0 Mon Sep 03 17:38:38 CEST 2012 WEB-INF/
     0 Mon Sep 03 17:38:38 CEST 2012 WEB-INF/classes/
     0 Mon Sep 03 16:35:50 CEST 2012 WEB-INF/classes/org/
     0 Mon Sep 03 16:35:50 CEST 2012 WEB-INF/classes/org/jboss/
     0 Mon Sep 03 16:35:50 CEST 2012 WEB-INF/classes/org/jboss/test/
     0 Mon Sep 03 16:35:52 CEST 2012 WEB-INF/classes/org/jboss/test/ws/
     0 Mon Sep 03 16:35:50 CEST 2012 WEB-INF/classes/org/jboss/test/ws/jaxws/
     0 Mon Sep 03 16:35:52 CEST 2012 WEB-INF/classes/org/jboss/test/ws/jaxws/samples/
     0 Mon Sep 03 16:35:50 CEST 2012 WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/
     0 Mon Sep 03 16:35:50 CEST 2012
WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/
     0 Mon Sep 03 16:35:52 CEST 2012
WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/trust/
  1686 Mon Sep 03 16:35:50 CEST 2012
WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/trust/PicketLinkSTService.class
  1148 Mon Sep 03 16:35:52 CEST 2012
WEB-INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/trust/STSCallbackHandler.class
   251 Mon Sep 03 17:38:34 CEST 2012 WEB-INF/jboss-web.xml
     0 Mon Sep 03 16:35:50 CEST 2012 WEB-INF/wsdl/
  9070 Mon Sep 03 17:38:34 CEST 2012 WEB-INF/wsdl/PicketLinkSTS.wsdl
  1267 Mon Sep 03 17:38:34 CEST 2012 WEB-INF/classes/picketlink-sts.xml
  1054 Mon Sep 03 16:35:50 CEST 2012 WEB-INF/classes/stsKeystore.properties
  3978 Mon Sep 03 16:35:50 CEST 2012 WEB-INF/classes/stsstore.jks
```

ActAs WS-Trust Scenario

ActAs WS-Trust Scenario

The ActAs feature is used in scenarios that require composite delegation. It is commonly used in multi-tiered systems where an application calls a service on behalf of a logged in user or a service calls another service on behalf of the original caller.

ActAs is nothing more than a new sub-element in the RequestSecurityToken (RST). It provides additional information about the original caller when a token is negotiated with the STS. The ActAs element usually takes the form of a token with identity claims such as name, role, and authorization code, for the client to access the service.

The ActAs scenario is an extension of the basic WS-Trust scenario. In this example the ActAs service calls the ws-service on behalf of a user. There are only a couple of additions to the basic scenario's code. An ActAs web service provider and callback handler have been added. The ActAs web services' WSDL imposes the same security policies as the ws-provider. UsernameTokenCallbackHandler is new. It is a utility that generates the content for the ActAs element. And lastly there are a couple of code additions in the STS to support the ActAs request.

Web service provider

This section examines the web service elements from the basic WS-Trust scenario that have been changed to address the needs of the ActAs example. The components are

- ActAs web service provider's WSDL
- ActAs web service provider's Interface and Implementation classes.
- ActAsCallbackHandler class
- UsernameTokenCallbackHandler
- Crypto properties and keystore files
- MANIFEST.MF

Web service provider WSDL

The ActAs web service provider's WSDL is a clone of the ws-provider's WSDL. The wsp:Policy section is the same. There are changes to the service endpoint, targetNamespace, portType, binding name, and service.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions targetNamespace="http://www.jboss.org/jbossws/ws-extensions/actaswssecuritypolicy"
name="ActAsService"
             xmlns:tns="http://www.jboss.org/jbossws/ws-extensions/actaswssecuritypolicy"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns="http://schemas.xmlsoap.org/wsdl/"
             xmlns:wsp="http://www.w3.org/ns/ws-policy"
             xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"

xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
             xmlns:wsaws="http://www.w3.org/2005/08/addressing"
             xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702"
             xmlns:t="http://docs.oasis-open.org/ws-sx/ws-trust/200512">
    <types>
        <xsd:schema>
            <xsd:import
namespace="http://www.jboss.org/jbossws/ws-extensions/actaswssecuritypolicy"
                    schemaLocation="ActAsService_schema1.xsd"/>
        </xsd:schema>
    </types>
    <message name="sayHello">
        <part name="parameters" element="tns:sayHello"/>
    </message>
    <message name="sayHelloResponse">
        <part name="parameters" element="tns:sayHelloResponse"/>
    </message>
    <portType name="ActAsServiceIface">
        <operation name="sayHello">
            <input message="tns:sayHello"/>
            <output message="tns:sayHelloResponse"/>
        </operation>
    </portType>
    <binding name="ActAsServicePortBinding" type="tns:ActAsServiceIface">
        <wsp:PolicyReference URI="#AsymmetricSAML2Policy" />
        <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
        <operation name="sayHello">
            <soap:operation soapAction=""/>
            <input>
                <soap:body use="literal"/>
                <wsp:PolicyReference URI="#Input_Policy" />
            </input>
            <output>
                <soap:body use="literal"/>
                <wsp:PolicyReference URI="#Output_Policy" />
            </output>
        </operation>
    </binding>
    <service name="ActAsService">
        <port name="ActAsServicePort" binding="tns:ActAsServicePortBinding">
            <soap:address
location="http://@jboss.bind.address@:8080/jaxws-samples-wsse-policy-trust-actas/ActAsService"/>
        </port>
    </service>

</definitions>
```

Web Service Interface

The web service provider interface class, ActAsServiceIface, is a simple web service definition.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.actas;

import javax.jws.WebMethod;
import javax.jws.WebService;

@WebService
(
   targetNamespace = "http://www.jboss.org/jbossws/ws-extensions/actaswssecuritypolicy"
)
public interface ActAsServiceIface
{
   @WebMethod
   String sayHello();
}
```

Web Service Implementation

The web service provider implementation class, ActAsServiceImpl, is a simple POJO. It uses the standard WebService annotation to define the service endpoint and two Apache WSS4J annotations, EndpointProperties and EndpointProperty used for configuring the endpoint for the CXF runtime. The WSS4J configuration information provided is for WSS4J's Crypto Merlin implementation.

ActAsServiceImpl is calling ServiceImpl acting on behalf of the user. Method setupService performs the requisite configuration setup.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.actas;

import org.apache.cxf.Bus;
import org.apache.cxf.BusFactory;
import org.apache.cxf.annotations.EndpointProperties;
import org.apache.cxf.annotations.EndpointProperty;
import org.apache.cxf.ws.security.SecurityConstants;
import org.apache.cxf.ws.security.trust.STSClient;
import org.jboss.test.ws.jaxws.samples.wsse.policy.trust.service.ServiceIface;
import org.jboss.test.ws.jaxws.samples.wsse.policy.trust.shared.WSTrustAppUtils;

import javax.jws.WebService;
import javax.xml.namespace.QName;
import javax.xml.ws.BindingProvider;
import javax.xml.ws.Service;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.Map;

@WebService
(
   portName = "ActAsServicePort",
   serviceName = "ActAsService",
   wsdlLocation = "WEB-INF/wsdl/ActAsService.wsdl",
   targetNamespace = "http://www.jboss.org/jbossws/ws-extensions/actaswssecuritypolicy",
```

```
   endpointInterface =
"org.jboss.test.ws.jaxws.samples.wsse.policy.trust.actas.ActAsServiceIface"
)

@EndpointProperties(value = {
      @EndpointProperty(key = "ws-security.signature.username", value = "myactaskey"),
      @EndpointProperty(key = "ws-security.signature.properties", value =
"actasKeystore.properties"),
      @EndpointProperty(key = "ws-security.encryption.properties", value =
"actasKeystore.properties"),
      @EndpointProperty(key = "ws-security.callback-handler", value =
"org.jboss.test.ws.jaxws.samples.wsse.policy.trust.actas.ActAsCallbackHandler")
})

public class ActAsServiceImpl implements ActAsServiceIface
{
   public String sayHello() {
      try {
         ServiceIface proxy = setupService();
         return "ActAs " + proxy.sayHello();
      } catch (MalformedURLException e) {
         e.printStackTrace();
      }
      return null;
   }

   private  ServiceIface setupService()throws MalformedURLException {
      ServiceIface proxy = null;
      Bus bus = BusFactory.newInstance().createBus();

      try {
         BusFactory.setThreadDefaultBus(bus);

         final String serviceURL = "http://" + WSTrustAppUtils.getServerHost() +
":8080/jaxws-samples-wsse-policy-trust/SecurityService";
         final QName serviceName = new
QName("http://www.jboss.org/jbossws/ws-extensions/wssecuritypolicy", "SecurityService");
         final URL wsdlURL = new URL(serviceURL + "?wsdl");
         Service service = Service.create(wsdlURL, serviceName);
         proxy = (ServiceIface) service.getPort(ServiceIface.class);

         Map<String, Object> ctx = ((BindingProvider) proxy).getRequestContext();
         ctx.put(SecurityConstants.CALLBACK_HANDLER, new ActAsCallbackHandler());

         ctx.put(SecurityConstants.SIGNATURE_PROPERTIES,

Thread.currentThread().getContextClassLoader().getResource("actasKeystore.properties" ));
         ctx.put(SecurityConstants.SIGNATURE_USERNAME, "myactaskey" );
         ctx.put(SecurityConstants.ENCRYPT_PROPERTIES,

Thread.currentThread().getContextClassLoader().getResource("../../META-INF/clientKeystore.properti
));
         ctx.put(SecurityConstants.ENCRYPT_USERNAME, "myservicekey");

         STSClient stsClient = new STSClient(bus);
         Map<String, Object> props = stsClient.getProperties();
         props.put(SecurityConstants.USERNAME, "alice");
         props.put(SecurityConstants.ENCRYPT_USERNAME, "mystskey");
```

```
         props.put(SecurityConstants.STS_TOKEN_USERNAME, "myactaskey" );
         props.put(SecurityConstants.STS_TOKEN_PROPERTIES,

Thread.currentThread().getContextClassLoader().getResource("actasKeystore.properties" ));
         props.put(SecurityConstants.STS_TOKEN_USE_CERT_FOR_KEYINFO, "true");

         ctx.put(SecurityConstants.STS_CLIENT, stsClient);

      } finally {
         bus.shutdown(true);
      }

      return proxy;
   }

}
```

ActAsCallbackHandler

ActAsCallbackHandler is a callback handler for the WSS4J Crypto API. It is used to obtain the password for the private key in the keystore. This class enables CXF to retrieve the password of the user name to use for the message signature. This class has been revised to return the passwords for this service, myactaskey and the "actas" user, alice.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.actas;

import org.jboss.wsf.stack.cxf.extensions.security.PasswordCallbackHandler;
import java.util.HashMap;
import java.util.Map;

public class ActAsCallbackHandler extends PasswordCallbackHandler {

   public ActAsCallbackHandler()
   {
      super(getInitMap());
   }

   private static Map<String, String> getInitMap()
   {
      Map<String, String> passwords = new HashMap<String, String>();
      passwords.put("myactaskey", "aspass");
      passwords.put("alice", "clarinet");
      return passwords;
   }
}
```

UsernameTokenCallbackHandler

The ActAs and OnBeholdOf sub-elements of the RequestSecurityToken are required to be defined as WSSE Username Tokens. This utility generates the properly formated element.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.shared;

import org.apache.cxf.helpers.DOMUtils;
```

```
import org.apache.cxf.message.Message;
import org.apache.cxf.ws.security.SecurityConstants;
import org.apache.cxf.ws.security.trust.delegation.DelegationCallback;
import org.apache.ws.security.WSConstants;
import org.apache.ws.security.message.token.UsernameToken;
import org.w3c.dom.Document;
import org.w3c.dom.Node;
import org.w3c.dom.Element;
import org.w3c.dom.ls.DOMImplementationLS;
import org.w3c.dom.ls.LSSerializer;

import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import java.io.IOException;
import java.util.Map;

/**
* A utility to provide the 3 different input parameter types for jaxws property
* "ws-security.sts.token.act-as" and "ws-security.sts.token.on-behalf-of".
* This implementation obtains a username and password via the jaxws property
* "ws-security.username" and "ws-security.password" respectively, as defined
* in SecurityConstants.  It creates a wss UsernameToken to be used as the
* delegation token.
*/

public class UsernameTokenCallbackHandler implements CallbackHandler {

   public void handle(Callback[] callbacks)
      throws IOException, UnsupportedCallbackException {
      for (int i = 0; i < callbacks.length; i++) {
         if (callbacks[i] instanceof DelegationCallback) {
            DelegationCallback callback = (DelegationCallback) callbacks[i];
            Message message = callback.getCurrentMessage();

            String username =
               (String)message.getContextualProperty(SecurityConstants.USERNAME);
            String password =
               (String)message.getContextualProperty(SecurityConstants.PASSWORD);
            if (username != null) {
               Node contentNode = message.getContent(Node.class);
               Document doc = null;
               if (contentNode != null) {
                  doc = contentNode.getOwnerDocument();
               } else {
                  doc = DOMUtils.createDocument();
               }
               UsernameToken usernameToken = createWSSEUsernameToken(username,password, doc);
               callback.setToken(usernameToken.getElement());
            }
         } else {
            throw new UnsupportedCallbackException(callbacks[i], "Unrecognized Callback");
         }
      }
   }

   /**
    * Provide UsernameToken as a string.
```

```
     * @param ctx
     * @return
     */
    public String getUsernameTokenString(Map<String, Object> ctx){
       Document doc = DOMUtils.createDocument();
       String result = null;
       String username = (String)ctx.get(SecurityConstants.USERNAME);
       String password = (String)ctx.get(SecurityConstants.PASSWORD);
       if (username != null) {
          UsernameToken usernameToken = createWSSEUsernameToken(username,password, doc);
          result = toString(usernameToken.getElement().getFirstChild().getParentNode());
       }
       return result;
    }

    /**
     *
     * @param username
     * @param password
     * @return
     */
    public String getUsernameTokenString(String username, String password){
       Document doc = DOMUtils.createDocument();
       String result = null;
       if (username != null) {
          UsernameToken usernameToken = createWSSEUsernameToken(username,password, doc);
          result = toString(usernameToken.getElement().getFirstChild().getParentNode());
       }
       return result;
    }

    /**
     * Provide UsernameToken as a DOM Element.
     * @param ctx
     * @return
     */
    public Element getUsernameTokenElement(Map<String, Object> ctx){
       Document doc = DOMUtils.createDocument();
       Element result = null;
       UsernameToken usernameToken = null;
          String username = (String)ctx.get(SecurityConstants.USERNAME);
       String password = (String)ctx.get(SecurityConstants.PASSWORD);
       if (username != null) {
          usernameToken = createWSSEUsernameToken(username,password, doc);
          result = usernameToken.getElement();
       }
       return result;
    }

    /**
     *
     * @param username
     * @param password
     * @return
     */
    public Element getUsernameTokenElement(String username, String password){
       Document doc = DOMUtils.createDocument();
       Element result = null;
```

```
      UsernameToken usernameToken = null;
      if (username != null) {
         usernameToken = createWSSEUsernameToken(username,password, doc);
         result = usernameToken.getElement();
      }
      return result;
   }

   private UsernameToken createWSSEUsernameToken(String username, String password, Document doc)
{

      UsernameToken usernameToken = new UsernameToken(true, doc,
         (password == null)? null: WSConstants.PASSWORD_TEXT);
      usernameToken.setName(username);
      usernameToken.addWSUNamespace();
      usernameToken.addWSSENamespace();
      usernameToken.setID("id-" + username);

      if (password != null){
         usernameToken.setPassword(password);
      }

      return usernameToken;
   }


   private String toString(Node node) {
      String str = null;

      if (node != null) {
         DOMImplementationLS lsImpl = (DOMImplementationLS)
            node.getOwnerDocument().getImplementation().getFeature("LS", "3.0");
         LSSerializer serializer = lsImpl.createLSSerializer();
         serializer.getDomConfig().setParameter("xml-declaration", false); //by default its
true, so set it to false to get String without xml-declaration
         str = serializer.writeToString(node);
      }
      return str;
   }

}
```

Crypto properties and keystore files

The ActAs service must provide its own credentials. The requisite properties file, actasKeystore.properties, and keystore, actasstore.jks, were created.

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=aapass
org.apache.ws.security.crypto.merlin.keystore.alias=myactaskey
org.apache.ws.security.crypto.merlin.keystore.file=actasstore.jks
```

MANIFEST.MF

When deployed on WildFly this application requires access to the JBossWs and CXF APIs provided in modules org.jboss.ws.cxf.jbossws-cxf-client and org.apache.cxf. The Apache CXF internals, org.apache.cxf.impl, are needed in handling the ActAs and OnBehalfOf extensions. The dependency statement directs the server to provide them at deployment.

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.8.2
Created-By: 1.7.0_25-b15 (Oracle Corporation)
Dependencies: org.jboss.ws.cxf.jbossws-cxf-client, org.apache.cxf.impl
```

Security Token Service

This section examines the STS elements from the basic WS-Trust scenario that have been changed to address the needs of the ActAs example. The components are.

- STS's implementation class.
- STSCallbackHandler class

STS Implementation class

The initial description of SampleSTS can be found here.

The declaration of the set of allowed token recipients by address has been extended to accept ActAs addresses and OnBehalfOf addresses. The addresses are specified as reg-ex patterns.

The TokenIssueOperation requires class, UsernameTokenValidator be provided in order to validate the contents of the OnBehalfOf claims and class, UsernameTokenDelegationHandler to be provided in order to process the token delegation request of the ActAs on OnBehalfOf user.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.sts;

import java.util.Arrays;
import java.util.LinkedList;
import java.util.List;

import javax.xml.ws.WebServiceProvider;

import org.apache.cxf.annotations.EndpointProperties;
import org.apache.cxf.annotations.EndpointProperty;
import org.apache.cxf.interceptor.InInterceptors;
import org.apache.cxf.sts.StaticSTSProperties;
import org.apache.cxf.sts.operation.TokenIssueOperation;
import org.apache.cxf.sts.operation.TokenValidateOperation;
import org.apache.cxf.sts.service.ServiceMBean;
import org.apache.cxf.sts.service.StaticService;
import org.apache.cxf.sts.token.delegation.UsernameTokenDelegationHandler;
import org.apache.cxf.sts.token.provider.SAMLTokenProvider;
import org.apache.cxf.sts.token.validator.SAMLTokenValidator;
import org.apache.cxf.sts.token.validator.UsernameTokenValidator;
import org.apache.cxf.ws.security.sts.provider.SecurityTokenServiceProvider;

@WebServiceProvider(serviceName = "SecurityTokenService",
```

```
      portName = "UT_Port",
      targetNamespace = "http://docs.oasis-open.org/ws-sx/ws-trust/200512/",
      wsdlLocation = "WEB-INF/wsdl/ws-trust-1.4-service.wsdl")
//be sure to have dependency on org.apache.cxf module when on AS7, otherwise Apache CXF
annotations are ignored
@EndpointProperties(value = {
      @EndpointProperty(key = "ws-security.signature.username", value = "mystskey"),
      @EndpointProperty(key = "ws-security.signature.properties", value =
"stsKeystore.properties"),
      @EndpointProperty(key = "ws-security.callback-handler", value =
"org.jboss.test.ws.jaxws.samples.wsse.policy.trust.sts.STSCallbackHandler"),
      @EndpointProperty(key = "ws-security.validate.token", value = "false") //to let the JAAS
integration deal with validation through the interceptor below
})
@InInterceptors(interceptors =
{"org.jboss.wsf.stack.cxf.security.authentication.SubjectCreatingPolicyInterceptor"})
public class SampleSTS extends SecurityTokenServiceProvider
{
   public SampleSTS() throws Exception
   {
      super();

      StaticSTSProperties props = new StaticSTSProperties();
      props.setSignatureCryptoProperties("stsKeystore.properties");
      props.setSignatureUsername("mystskey");
      props.setCallbackHandlerClass(STSCallbackHandler.class.getName());
      props.setIssuer("DoubleItSTSIssuer");

      List<ServiceMBean> services = new LinkedList<ServiceMBean>();
      StaticService service = new StaticService();
      service.setEndpoints(Arrays.asList(
         "http://localhost:(\\d)*/jaxws-samples-wsse-policy-trust/SecurityService",
         "http://\\[::1\\]:(\\d)*/jaxws-samples-wsse-policy-trust/SecurityService",
         "http://\\[0:0:0:0:0:0:0:1\\]:(\\d)*/jaxws-samples-wsse-policy-trust/SecurityService",

         "http://localhost:(\\d)*/jaxws-samples-wsse-policy-trust-actas/ActAsService",
         "http://\\[::1\\]:(\\d)*/jaxws-samples-wsse-policy-trust-actas/ActAsService",

"http://\\[0:0:0:0:0:0:0:1\\]:(\\d)*/jaxws-samples-wsse-policy-trust-actas/ActAsService",

         "http://localhost:(\\d)*/jaxws-samples-wsse-policy-trust-onbehalfof/OnBehalfOfService",
         "http://\\[::1\\]:(\\d)*/jaxws-samples-wsse-policy-trust-onbehalfof/OnBehalfOfService",

"http://\\[0:0:0:0:0:0:0:1\\]:(\\d)*/jaxws-samples-wsse-policy-trust-onbehalfof/OnBehalfOfService"
));
      services.add(service);

      TokenIssueOperation issueOperation = new TokenIssueOperation();
      issueOperation.setServices(services);
      issueOperation.getTokenProviders().add(new SAMLTokenProvider());
      // required for OnBehalfOf
      issueOperation.getTokenValidators().add(new UsernameTokenValidator());
      // added for OnBehalfOf and ActAs
      issueOperation.getDelegationHandlers().add(new UsernameTokenDelegationHandler());
      issueOperation.setStsProperties(props);

      TokenValidateOperation validateOperation = new TokenValidateOperation();
      validateOperation.getTokenValidators().add(new SAMLTokenValidator());
```

```
      validateOperation.setStsProperties(props);

      this.setIssueOperation(issueOperation);
      this.setValidateOperation(validateOperation);
   }
}
```

STSCallbackHandler

The user, alice, and corresponding password was required to be added for the ActAs example.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.sts;

import java.util.HashMap;
import java.util.Map;

import org.jboss.wsf.stack.cxf.extensions.security.PasswordCallbackHandler;

public class STSCallbackHandler extends PasswordCallbackHandler
{
   public STSCallbackHandler()
   {
      super(getInitMap());
   }

   private static Map<String, String> getInitMap()
   {
      Map<String, String> passwords = new HashMap<String, String>();
      passwords.put("mystskey", "stskpass");
      passwords.put("alice", "clarinet");
      return passwords;
   }
}
```

Web service requester

This section examines the ws-requester elements from the basic WS-Trust scenario that have been changed to address the needs of the ActAs example. The component is

- ActAs web service requester implementation class

Web service requester Implementation

The ActAs ws-requester, the client, uses standard procedures for creating a reference to the web service in the first four lines. To address the endpoint security requirements, the web service's "Request Context" is configured via the BindingProvider. Information needed in the message generation is provided through it. The ActAs user, myactaskey, is declared in this section and UsernameTokenCallbackHandler is used to provide the contents of the ActAs element to the STSClient. In this example a STSClient object is created and provided to the proxy's request context. The alternative is to provide keys tagged with the ".it" suffix as was done in the Basic Scenario client. The use of ActAs is configured through the props map using the SecurityConstants.STS_TOKEN_ACT_AS key. The alternative is to use the STSClient.setActAs method.

```
final QName serviceName = new
QName("http://www.jboss.org/jbossws/ws-extensions/actaswssecuritypolicy", "ActAsService");
final URL wsdlURL = new URL(serviceURL + "?wsdl");
Service service = Service.create(wsdlURL, serviceName);
ActAsServiceIface proxy = (ActAsServiceIface) service.getPort(ActAsServiceIface.class);

Bus bus = BusFactory.newInstance().createBus();
try {
    BusFactory.setThreadDefaultBus(bus);

    Map<String, Object> ctx = proxy.getRequestContext();

    ctx.put(SecurityConstants.CALLBACK_HANDLER, new ClientCallbackHandler());
    ctx.put(SecurityConstants.ENCRYPT_PROPERTIES,
        Thread.currentThread().getContextClassLoader().getResource(
        "META-INF/clientKeystore.properties"));
    ctx.put(SecurityConstants.ENCRYPT_USERNAME, "myactaskey");
    ctx.put(SecurityConstants.SIGNATURE_PROPERTIES,
        Thread.currentThread().getContextClassLoader().getResource(
        "META-INF/clientKeystore.properties"));
    ctx.put(SecurityConstants.SIGNATURE_USERNAME, "myclientkey");

    // Generate the ActAs element contents and pass to the STSClient as a string
    UsernameTokenCallbackHandler ch = new UsernameTokenCallbackHandler();
    String str = ch.getUsernameTokenString("alice","clarinet");
    ctx.put(SecurityConstants.STS_TOKEN_ACT_AS, str);

    STSClient stsClient = new STSClient(bus);
    Map<String, Object> props = stsClient.getProperties();
    props.put(SecurityConstants.USERNAME, "bob");
    props.put(SecurityConstants.CALLBACK_HANDLER, new ClientCallbackHandler());
    props.put(SecurityConstants.ENCRYPT_PROPERTIES,
        Thread.currentThread().getContextClassLoader().getResource(
        "META-INF/clientKeystore.properties"));
    props.put(SecurityConstants.ENCRYPT_USERNAME, "mystskey");
    props.put(SecurityConstants.STS_TOKEN_USERNAME, "myclientkey");
    props.put(SecurityConstants.STS_TOKEN_PROPERTIES,
        Thread.currentThread().getContextClassLoader().getResource(
        "META-INF/clientKeystore.properties"));
    props.put(SecurityConstants.STS_TOKEN_USE_CERT_FOR_KEYINFO, "true");

    ctx.put(SecurityConstants.STS_CLIENT, stsClient);
} finally {
    bus.shutdown(true);
}
proxy.sayHello();
```

OnBehalfOf WS-Trust Scenario

OnBehalfOf WS-Trust Scenario

The OnBehalfOf feature is used in scenarios that use the proxy pattern. In such scenarios, the client cannot access the STS directly, instead it communicates through a proxy gateway. The proxy gateway authenticates the caller and puts information about the caller into the OnBehalfOf element of the RequestSecurityToken (RST) sent to the real STS for processing. The resulting token contains only claims related to the client of the proxy, making the proxy completely transparent to the receiver of the issued token.

OnBehalfOf is nothing more than a new sub-element in the RST. It provides additional information about the original caller when a token is negotiated with the STS. The OnBehalfOf element usually takes the form of a token with identity claims such as name, role, and authorization code, for the client to access the service.

The OnBehalfOf scenario is an extension of the basic WS-Trust scenario. In this example the OnBehalfOf service calls the ws-service on behalf of a user. There are only a couple of additions to the basic scenario's code. An OnBehalfOf web service provider and callback handler have been added. The OnBehalfOf web services' WSDL imposes the same security policies as the ws-provider. UsernameTokenCallbackHandler is a utility shared with ActAs. It generates the content for the OnBehalfOf element. And lastly there are code additions in the STS that both OnBehalfOf and ActAs share in common.

Infor here [Open Source Security: Apache CXF 2.5.1 STS updates]

Web service provider
This section examines the web service elements from the basic WS-Trust scenario that have been changed to address the needs of the OnBehalfOf example. The components are.

- web service provider's WSDL
- web service provider's Interface and Implementation classes.
- OnBehalfOfCallbackHandler class

Web service provider WSDL
The OnBehalfOf web service provider's WSDL is a clone of the ws-provider's WSDL. The wsp:Policy section is the same. There are changes to the service endpoint, targetNamespace, portType, binding name, and service.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions
targetNamespace="http://www.jboss.org/jbossws/ws-extensions/onbehalfofwssecuritypolicy"
name="OnBehalfOfService"
             xmlns:tns="http://www.jboss.org/jbossws/ws-extensions/onbehalfofwssecuritypolicy"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns="http://schemas.xmlsoap.org/wsdl/"
             xmlns:wsp="http://www.w3.org/ns/ws-policy"
             xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"

xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
             xmlns:wsaws="http://www.w3.org/2005/08/addressing"
             xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702"
             xmlns:t="http://docs.oasis-open.org/ws-sx/ws-trust/200512">
    <types>
        <xsd:schema>
            <xsd:import
namespace="http://www.jboss.org/jbossws/ws-extensions/onbehalfofwssecuritypolicy"
                  schemaLocation="OnBehalfOfService_schema1.xsd"/>
        </xsd:schema>
    </types>
    <message name="sayHello">
        <part name="parameters" element="tns:sayHello"/>
    </message>
    <message name="sayHelloResponse">
        <part name="parameters" element="tns:sayHelloResponse"/>
    </message>
    <portType name="OnBehalfOfServiceIface">
        <operation name="sayHello">
            <input message="tns:sayHello"/>
            <output message="tns:sayHelloResponse"/>
        </operation>
    </portType>
    <binding name="OnBehalfOfServicePortBinding" type="tns:OnBehalfOfServiceIface">
        <wsp:PolicyReference URI="#AsymmetricSAML2Policy" />
        <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
        <operation name="sayHello">
            <soap:operation soapAction=""/>
            <input>
                <soap:body use="literal"/>
                <wsp:PolicyReference URI="#Input_Policy" />
            </input>
            <output>
                <soap:body use="literal"/>
                <wsp:PolicyReference URI="#Output_Policy" />
            </output>
        </operation>
    </binding>
    <service name="OnBehalfOfService">
        <port name="OnBehalfOfServicePort" binding="tns:OnBehalfOfServicePortBinding">
            <soap:address
location="http://@jboss.bind.address@:8080/jaxws-samples-wsse-policy-trust-onbehalfof/OnBehalfOfSe
</port>
    </service>
</definitions>
```

Web Service Interface

The web service provider interface class, OnBehalfOfServiceIface, is a simple web service definition.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.onbehalfof;

import javax.jws.WebMethod;
import javax.jws.WebService;

@WebService
(
   targetNamespace = "http://www.jboss.org/jbossws/ws-extensions/onbehalfofwssecuritypolicy"
)
public interface OnBehalfOfServiceIface
{
   @WebMethod
   String sayHello();
}
```

Web Service Implementation

The web service provider implementation class, OnBehalfOfServiceImpl, is a simple POJO. It uses the standard WebService annotation to define the service endpoint and two Apache WSS4J annotations, EndpointProperties and EndpointProperty used for configuring the endpoint for the CXF runtime. The WSS4J configuration information provided is for WSS4J's Crypto Merlin implementation.

OnBehalfOfServiceImpl is calling the ServiceImpl acting on behalf of the user. Method setupService performs the requisite configuration setup.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.onbehalfof;

import org.apache.cxf.Bus;
import org.apache.cxf.BusFactory;
import org.apache.cxf.annotations.EndpointProperties;
import org.apache.cxf.annotations.EndpointProperty;
import org.apache.cxf.ws.security.SecurityConstants;
import org.apache.cxf.ws.security.trust.STSClient;
import org.jboss.test.ws.jaxws.samples.wsse.policy.trust.service.ServiceIface;
import org.jboss.test.ws.jaxws.samples.wsse.policy.trust.shared.WSTrustAppUtils;

import javax.jws.WebService;
import javax.xml.namespace.QName;
import javax.xml.ws.BindingProvider;
import javax.xml.ws.Service;
import java.net.*;
import java.util.Map;

@WebService
(
   portName = "OnBehalfOfServicePort",
   serviceName = "OnBehalfOfService",
   wsdlLocation = "WEB-INF/wsdl/OnBehalfOfService.wsdl",
   targetNamespace = "http://www.jboss.org/jbossws/ws-extensions/onbehalfofwssecuritypolicy",
   endpointInterface =
```

```
"org.jboss.test.ws.jaxws.samples.wsse.policy.trust.onbehalfof.OnBehalfOfServiceIface"
)

@EndpointProperties(value = {
      @EndpointProperty(key = "ws-security.signature.username", value = "myactaskey"),
      @EndpointProperty(key = "ws-security.signature.properties", value =
"actasKeystore.properties"),
      @EndpointProperty(key = "ws-security.encryption.properties", value =
"actasKeystore.properties"),
      @EndpointProperty(key = "ws-security.callback-handler", value =
"org.jboss.test.ws.jaxws.samples.wsse.policy.trust.onbehalfof.OnBehalfOfCallbackHandler")
})

public class OnBehalfOfServiceImpl implements OnBehalfOfServiceIface
{
   public String sayHello() {
      try {

         ServiceIface proxy = setupService();
         return "OnBehalfOf " + proxy.sayHello();

      } catch (MalformedURLException e) {
         e.printStackTrace();
      }
      return null;
   }

   /**
    *
    * @return
    * @throws MalformedURLException
    */
   private  ServiceIface setupService()throws MalformedURLException {
      ServiceIface proxy = null;
      Bus bus = BusFactory.newInstance().createBus();

      try {
         BusFactory.setThreadDefaultBus(bus);

         final String serviceURL = "http://" + WSTrustAppUtils.getServerHost() +
":8080/jaxws-samples-wsse-policy-trust/SecurityService";
         final QName serviceName = new
QName("http://www.jboss.org/jbossws/ws-extensions/wssecuritypolicy", "SecurityService");
         final URL wsdlURL = new URL(serviceURL + "?wsdl");
         Service service = Service.create(wsdlURL, serviceName);
         proxy = (ServiceIface) service.getPort(ServiceIface.class);

         Map<String, Object> ctx = ((BindingProvider) proxy).getRequestContext();
         ctx.put(SecurityConstants.CALLBACK_HANDLER, new OnBehalfOfCallbackHandler());

         ctx.put(SecurityConstants.SIGNATURE_PROPERTIES,
            Thread.currentThread().getContextClassLoader().getResource(
            "actasKeystore.properties" ));
         ctx.put(SecurityConstants.SIGNATURE_USERNAME, "myactaskey" );
         ctx.put(SecurityConstants.ENCRYPT_PROPERTIES,
            Thread.currentThread().getContextClassLoader().getResource(
            "../../META-INF/clientKeystore.properties" ));
         ctx.put(SecurityConstants.ENCRYPT_USERNAME, "myservicekey");
```

```
         STSClient stsClient = new STSClient(bus);
         Map<String, Object> props = stsClient.getProperties();
         props.put(SecurityConstants.USERNAME, "bob");
         props.put(SecurityConstants.ENCRYPT_USERNAME, "mystskey");
         props.put(SecurityConstants.STS_TOKEN_USERNAME, "myactaskey" );
         props.put(SecurityConstants.STS_TOKEN_PROPERTIES,
            Thread.currentThread().getContextClassLoader().getResource(
            "actasKeystore.properties" ));
         props.put(SecurityConstants.STS_TOKEN_USE_CERT_FOR_KEYINFO, "true");

         ctx.put(SecurityConstants.STS_CLIENT, stsClient);

      } finally {
         bus.shutdown(true);
      }

      return proxy;
   }

}
```

OnBehalfOfCallbackHandler

OnBehalfOfCallbackHandler is a callback handler for the WSS4J Crypto API. It is used to obtain the password for the private key in the keystore. This class enables CXF to retrieve the password of the user name to use for the message signature. This class has been revised to return the passwords for this service, myactaskey and the "OnBehalfOf" user, alice.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.onbehalfof;

import org.jboss.wsf.stack.cxf.extensions.security.PasswordCallbackHandler;
import java.util.HashMap;
import java.util.Map;

public class OnBehalfOfCallbackHandler extends PasswordCallbackHandler {

   public OnBehalfOfCallbackHandler()
   {
      super(getInitMap());
   }

   private static Map<String, String> getInitMap()
   {
      Map<String, String> passwords = new HashMap<String, String>();
      passwords.put("myactaskey", "aspass");
      passwords.put("alice", "clarinet");
      passwords.put("bob", "trombone");
      return passwords;
   }

}
```

Web service requester

This section examines the ws-requester elements from the basic WS-Trust scenario that have been changed to address the needs of the OnBehalfOf example. The component is

- OnBehalfOf web service requester implementation class

Web service requester Implementation

The OnBehalfOf ws-requester, the client, uses standard procedures for creating a reference to the web service in the first four lines. To address the endpoint security requirements, the web service's "Request Context" is configured via the BindingProvider. Information needed in the message generation is provided through it. The OnBehalfOf user, alice, is declared in this section and the callbackHandler, UsernameTokenCallbackHandler is provided to the STSClient for generation of the contents for the OnBehalfOf message element. In this example a STSClient object is created and provided to the proxy's request context. The alternative is to provide keys tagged with the ".it" suffix as was done in the Basic Scenario client . The use of OnBehalfOf is configured by the method call stsClient.setOnBehalfOf. The alternative is to use the key SecurityConstants.STS_TOKEN_ON_BEHALF_OF and a value in the props map.

```
final QName serviceName = new
QName("http://www.jboss.org/jbossws/ws-extensions/onbehalfofwssecuritypolicy",
"OnBehalfOfService");
final URL wsdlURL = new URL(serviceURL + "?wsdl");
Service service = Service.create(wsdlURL, serviceName);
OnBehalfOfServiceIface proxy = (OnBehalfOfServiceIface)
service.getPort(OnBehalfOfServiceIface.class);


Bus bus = BusFactory.newInstance().createBus();
try {

    BusFactory.setThreadDefaultBus(bus);

    Map<String, Object> ctx = proxy.getRequestContext();

    ctx.put(SecurityConstants.CALLBACK_HANDLER, new ClientCallbackHandler());
    ctx.put(SecurityConstants.ENCRYPT_PROPERTIES,
        Thread.currentThread().getContextClassLoader().getResource(
        "META-INF/clientKeystore.properties"));
    ctx.put(SecurityConstants.ENCRYPT_USERNAME, "myactaskey");
    ctx.put(SecurityConstants.SIGNATURE_PROPERTIES,
        Thread.currentThread().getContextClassLoader().getResource(
        "META-INF/clientKeystore.properties"));
    ctx.put(SecurityConstants.SIGNATURE_USERNAME, "myclientkey");

    // user and password OnBehalfOf user
    // UsernameTokenCallbackHandler will extract this information when called
    ctx.put(SecurityConstants.USERNAME,"alice");
    ctx.put(SecurityConstants.PASSWORD, "clarinet");

    STSClient stsClient = new STSClient(bus);

    // Providing the STSClient the mechanism to create the claims contents for OnBehalfOf
    stsClient.setOnBehalfOf(new UsernameTokenCallbackHandler());

    Map<String, Object> props = stsClient.getProperties();
    props.put(SecurityConstants.CALLBACK_HANDLER, new ClientCallbackHandler());
    props.put(SecurityConstants.ENCRYPT_PROPERTIES,
        Thread.currentThread().getContextClassLoader().getResource(
        "META-INF/clientKeystore.properties"));
    props.put(SecurityConstants.ENCRYPT_USERNAME, "mystskey");
    props.put(SecurityConstants.STS_TOKEN_USERNAME, "myclientkey");
    props.put(SecurityConstants.STS_TOKEN_PROPERTIES,
        Thread.currentThread().getContextClassLoader().getResource(
        "META-INF/clientKeystore.properties"));
    props.put(SecurityConstants.STS_TOKEN_USE_CERT_FOR_KEYINFO, "true");

    ctx.put(SecurityConstants.STS_CLIENT, stsClient);

} finally {
    bus.shutdown(true);
}
proxy.sayHello();
```

SAML Bearer Assertion Scenario

SAML Bearer Assertion Scenario

WS-Trust deals with managing software security tokens. A SAML assertion is a type of security token. In the SAML Bearer scenario, the service provider automatically trusts that the incoming SOAP request came from the subject defined in the SAML token after the service verifies the tokens signature.

Implementation of this scenario has the following requirements.

- SAML tokens with a Bearer subject confirmation method must be protected so the token can not be snooped. In most cases, a bearer token combined with HTTPS is sufficient to prevent "a man in the middle" getting possession of the token. This means a security policy that uses a sp:TransportBinding and sp:HttpsToken.
- A bearer token has no encryption or signing keys associated with it, therefore a sp:IssuedToken of bearer keyType should be used with a sp:SupportingToken or a sp:SignedSupportingTokens.

Web service Provider
This section examines the web service elements for the SAML Bearer scenario. The components are

- Bearer web service provider's WSDL
- SSL configuration
- Bearer web service provider's Interface and Implementation classes.
- Crypto properties and keystore files
- MANIFEST.MF

Web service provider WSDL

The web service provider is a contract-first endpoint. All the WS-trust and security policies for it are declared in WSDL, BearerService.wsdl. For this scenario a ws-requester is required to present a SAML 2.0 Bearer token issued from a designed STS. The address of the STS is provided in the WSDL. HTTPS, a TransportBinding and HttpsToken policy are used to protect the SOAP body of messages that pass back and forth between ws-requester and ws-provider. A detailed explanation of the security settings are provided in the comments in the listing below.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions targetNamespace="http://www.jboss.org/jbossws/ws-extensions/bearerwssecuritypolicy"
             name="BearerService"
             xmlns:tns="http://www.jboss.org/jbossws/ws-extensions/bearerwssecuritypolicy"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns="http://schemas.xmlsoap.org/wsdl/"
             xmlns:wsp="http://www.w3.org/ns/ws-policy"
             xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"

xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
             xmlns:wsaws="http://www.w3.org/2005/08/addressing"
             xmlns:wsx="http://schemas.xmlsoap.org/ws/2004/09/mex"
             xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702"
             xmlns:t="http://docs.oasis-open.org/ws-sx/ws-trust/200512">

  <types>
    <xsd:schema>
      <xsd:import namespace="http://www.jboss.org/jbossws/ws-extensions/bearerwssecuritypolicy"
                  schemaLocation="BearerService_schema1.xsd"/>
    </xsd:schema>
  </types>
  <message name="sayHello">
    <part name="parameters" element="tns:sayHello"/>
  </message>
  <message name="sayHelloResponse">
    <part name="parameters" element="tns:sayHelloResponse"/>
  </message>
  <portType name="BearerIface">
    <operation name="sayHello">
      <input message="tns:sayHello"/>
      <output message="tns:sayHelloResponse"/>
    </operation>
  </portType>

<!--
        The wsp:PolicyReference binds the security requirments on all the endpoints.
        The wsp:Policy wsu:Id="#TransportSAML2BearerPolicy" element is defined later in this
file.
-->
  <binding name="BearerServicePortBinding" type="tns:BearerIface">
    <wsp:PolicyReference URI="#TransportSAML2BearerPolicy" />
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
    <operation name="sayHello">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="literal"/>
```

```
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>

<!--
  The soap:address has been defined to use JBoss's https port, 8443.  This is
  set in conjunction with the sp:TransportBinding policy for https.
-->
  <service name="BearerService">
    <port name="BearerServicePort" binding="tns:BearerServicePortBinding">
      <soap:address
location="https://@jboss.bind.address@:8443/jaxws-samples-wsse-policy-trust-bearer/BearerService"/
</port>
  </service>


  <wsp:Policy wsu:Id="TransportSAML2BearerPolicy">
    <wsp:ExactlyOne>
      <wsp:All>
  <!--
        The wsam:Addressing element, indicates that the endpoints of this
        web service MUST conform to the WS-Addressing specification.  The
        attribute wsp:Optional="false" enforces this assertion.
  -->
        <wsam:Addressing wsp:Optional="false">
          <wsp:Policy />
        </wsam:Addressing>

<!--
  The sp:TransportBinding element indicates that security is provided by the
  message exchange transport medium, https.  WS-Security policy specification
  defines the sp:HttpsToken for use in exchanging messages transmitted over HTTPS.
-->
        <sp:TransportBinding
          xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
          <wsp:Policy>
            <sp:TransportToken>
              <wsp:Policy>
                <sp:HttpsToken>
                  <wsp:Policy/>
                </sp:HttpsToken>
              </wsp:Policy>
            </sp:TransportToken>
<!--
     The sp:AlgorithmSuite element, requires the TripleDes algorithm suite
     be used in performing cryptographic operations.
-->
            <sp:AlgorithmSuite>
              <wsp:Policy>
                <sp:TripleDes />
              </wsp:Policy>
            </sp:AlgorithmSuite>
<!--
     The sp:Layout element,  indicates the layout rules to apply when adding
     items to the security header.  The sp:Lax sub-element indicates items
```

```
     are added to the security header in any order that conforms to
     WSS: SOAP Message Security.
-->
             <sp:Layout>
               <wsp:Policy>
                 <sp:Lax />
               </wsp:Policy>
             </sp:Layout>
             <sp:IncludeTimestamp />
           </wsp:Policy>
        </sp:TransportBinding>

<!--
  The sp:SignedSupportingTokens element causes the supporting tokens
  to be signed using the primary token that is used to sign the message.
-->
        <sp:SignedSupportingTokens
          xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
          <wsp:Policy>
<!--
  The sp:IssuedToken element asserts that a SAML 2.0 security token of type
  Bearer is expected from the STS.  The

sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/AlwaysToRe
attribute instructs the runtime to include the initiator's public key
  with every message sent to the recipient.

  The sp:RequestSecurityTokenTemplate element directs that all of the
  children of this element will be copied directly into the body of the
  RequestSecurityToken (RST) message that is sent to the STS when the
  initiator asks the STS to issue a token.
-->
             <sp:IssuedToken

sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/AlwaysToRe
<sp:RequestSecurityTokenTemplate>

<t:TokenType>http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV2.0</t:TokenType
<t:KeyType>http://docs.oasis-open.org/ws-sx/ws-trust/200512/Bearer</t:KeyType>
               </sp:RequestSecurityTokenTemplate>
               <wsp:Policy>
                 <sp:RequireInternalReference />
               </wsp:Policy>
<!--
  The sp:Issuer element defines the STS's address and endpoint information
  This information is used by the STSClient.
-->
               <sp:Issuer>

<wsaws:Address>http://@jboss.bind.address@:8080/jaxws-samples-wsse-policy-trust-sts-bearer/Securit
<wsaws:Metadata
                   xmlns:wsdli="http://www.w3.org/2006/01/wsdl-instance"

wsdli:wsdlLocation="http://@jboss.bind.address@:8080/jaxws-samples-wsse-policy-trust-sts-bearer/Se
<wsaw:ServiceName
                     xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl"
                     xmlns:stsns="http://docs.oasis-open.org/ws-sx/ws-trust/200512/"
                     EndpointName="UT_Port">stsns:SecurityTokenService</wsaw:ServiceName>
```

```
                </wsaws:Metadata>
              </sp:Issuer>

            </sp:IssuedToken>
          </wsp:Policy>
        </sp:SignedSupportingTokens>
<!--
    The sp:Wss11 element declares WSS: SOAP Message Security 1.1 options
    to be supported by the STS.  These particular elements generally refer
    to how keys are referenced within the SOAP envelope.  These are normally
    handled by CXF.
-->
        <sp:Wss11>
          <wsp:Policy>
            <sp:MustSupportRefIssuerSerial />
            <sp:MustSupportRefThumbprint />
            <sp:MustSupportRefEncryptedKey />
          </wsp:Policy>
        </sp:Wss11>
<!--
    The sp:Trust13 element declares controls for WS-Trust 1.3 options.
    They are policy assertions related to exchanges specifically with
    client and server challenges and entropy behaviors.  Again these are
    normally handled by CXF.
-->
        <sp:Trust13>
          <wsp:Policy>
            <sp:MustSupportIssuedTokens />
            <sp:RequireClientEntropy />
            <sp:RequireServerEntropy />
          </wsp:Policy>
        </sp:Trust13>
      </wsp:All>
    </wsp:ExactlyOne>
  </wsp:Policy>

</definitions>
```

SSL configuration

This web service is using https, therefore the JBoss server must be configured to provide SSL support in the Web subsystem. There are 2 components to SSL configuration.

- create a certificate keystore
- declare an SSL connector in the Web subsystem of the JBoss server configuration file.

Follow the directions in the, "*Using the pure Java implementation supplied by JSSE*" section in the SSL Setup Guide.

Here is an example of an SSL connector declaration.

```
<subsystem xmlns="urn:jboss:domain:web:1.4" default-virtual-server="default-host"
native="false">
  .....
  <connector name="jbws-https-connector" protocol="HTTP/1.1" scheme="https"
socket-binding="https" secure="true" enabled="true">
    <ssl key-alias="tomcat" password="changeit"
certificate-key-file="/myJbossHome/security/test.keystore" verify-client="false"/>
  </connector>
  ...
```

Web service Interface

The web service provider interface class, BearerIface, is a simple straight forward web service definition.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.bearer;

import javax.jws.WebMethod;
import javax.jws.WebService;

@WebService
(
   targetNamespace = "http://www.jboss.org/jbossws/ws-extensions/bearerwssecuritypolicy"
)
public interface BearerIface
{
   @WebMethod
   String sayHello();
}
```

Web service Implementation

The web service provider implementation class, BearerImpl, is a simple POJO. It uses the standard WebService annotation to define the service endpoint. In addition there are two Apache CXF annotations, EndpointProperties and EndpointProperty used for configuring the endpoint for the CXF runtime. These annotations come from the Apache WSS4J project, which provides a Java implementation of the primary WS-Security standards for Web Services. These annotations are programmatically adding properties to the endpoint. With plain Apache CXF, these properties are often set via the <jaxws:properties> element on the <jaxws:endpoint> element in the Spring config; these annotations allow the properties to be configured in the code.

WSS4J uses the Crypto interface to get keys and certificates for signature creation/verification, as is asserted by the WSDL for this service. The WSS4J configuration information being provided by BearerImpl is for Crypto's Merlin implementation. More information will be provided about this in the keystore section.

Because the web service provider automatically trusts that the incoming SOAP request came from the subject defined in the SAML token there is no need for a Crypto callbackHandler class or a signature username, unlike in prior examples, however in order to verify the message signature, the Java properties file that contains the (Merlin) crypto configuration information is still required.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.bearer;

import org.apache.cxf.annotations.EndpointProperties;
import org.apache.cxf.annotations.EndpointProperty;

import javax.jws.WebService;

@WebService
(
   portName = "BearerServicePort",
   serviceName = "BearerService",
   wsdlLocation = "WEB-INF/wsdl/BearerService.wsdl",
   targetNamespace = "http://www.jboss.org/jbossws/ws-extensions/bearerwssecuritypolicy",
   endpointInterface = "org.jboss.test.ws.jaxws.samples.wsse.policy.trust.bearer.BearerIface"
)
@EndpointProperties(value = {
   @EndpointProperty(key = "ws-security.signature.properties", value =
"serviceKeystore.properties")
})
public class BearerImpl implements BearerIface
{
   public String sayHello()
   {
      return "Bearer WS-Trust Hello World!";
   }
}
```

Crypto properties and keystore files

WSS4J's Crypto implementation is loaded and configured via a Java properties file that contains Crypto configuration data. The file contains implementation-specific properties such as a keystore location, password, default alias and the like. This application is using the Merlin implementation. File serviceKeystore.properties contains this information.

File servicestore.jks, is a Java KeyStore (JKS) repository. It contains self signed certificates for myservicekey and mystskey. *Self signed certificates are not appropriate for production use.*

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=sspass
org.apache.ws.security.crypto.merlin.keystore.alias=myservicekey
org.apache.ws.security.crypto.merlin.keystore.file=servicestore.jks
```

MANIFEST.MF

When deployed on WildFly this application requires access to the JBossWs and CXF APIs provided in module org.jboss.ws.cxf.jbossws-cxf-client. The dependency statement directs the server to provide them at deployment.

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.8.2
Created-By: 1.7.0_25-b15 (Oracle Corporation)
Dependencies: org.jboss.ws.cxf.jbossws-cxf-client
```

Bearer Security Token Service

This section examines the crucial elements in providing the Security Token Service functionality for providing a SAML Bearer token. The components that will be discussed are.

- Security Domain
- STS's WSDL
- STS's implementation class
- STSBearerCallbackHandler
- Crypto properties and keystore files
- MANIFEST.MF

Security Domain

The STS requires a JBoss security domain be configured. The jboss-web.xml descriptor declares a named security domain,"JBossWS-trust-sts" to be used by this service for authentication. This security domain requires two properties files and the addition of a security-domain declaration in the JBoss server configuration file.

For this scenario the domain needs to contain user *alice*, password *clarinet*, and role *friend*. See the listings below for jbossws-users.properties and jbossws-roles.properties. In addition the following XML must be added to the JBoss security subsystem in the server configuration file. Replace "**SOME_PATH**" with appropriate information.

```
<security-domain name="JBossWS-trust-sts">
  <authentication>
    <login-module code="UsersRoles" flag="required">
      <module-option name="usersProperties" value="/SOME_PATH/jbossws-users.properties"/>
      <module-option name="unauthenticatedIdentity" value="anonymous"/>
      <module-option name="rolesProperties" value="/SOME_PATH/jbossws-roles.properties"/>
    </login-module>
  </authentication>
</security-domain>
```

jboss-web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jboss-web PUBLIC "-//JBoss//DTD Web Application 2.4//EN" ">
<jboss-web>
  <security-domain>java:/jaas/JBossWS-trust-sts</security-domain>
</jboss-web>
```

jbossws-users.properties

```
# A sample users.properties file for use with the UsersRolesLoginModule
alice=clarinet
```

jbossws-roles.properties

```
# A sample roles.properties file for use with the UsersRolesLoginModule
alice=friend
```

STS's WSDL

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
  targetNamespace="http://docs.oasis-open.org/ws-sx/ws-trust/200512/"
  xmlns:tns="http://docs.oasis-open.org/ws-sx/ws-trust/200512/"
  xmlns:wstrust="http://docs.oasis-open.org/ws-sx/ws-trust/200512/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
```

```
xmlns:wsap10="http://www.w3.org/2006/05/addressing/wsdl"
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
xmlns:wsp="http://www.w3.org/ns/ws-policy"
xmlns:wst="http://docs.oasis-open.org/ws-sx/ws-trust/200512"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata">

<wsdl:types>
  <xs:schema elementFormDefault="qualified"
             targetNamespace='http://docs.oasis-open.org/ws-sx/ws-trust/200512'>

    <xs:element name='RequestSecurityToken'
                type='wst:AbstractRequestSecurityTokenType'/>
    <xs:element name='RequestSecurityTokenResponse'
                type='wst:AbstractRequestSecurityTokenType'/>

    <xs:complexType name='AbstractRequestSecurityTokenType'>
      <xs:sequence>
        <xs:any namespace='##any' processContents='lax' minOccurs='0'
                maxOccurs='unbounded'/>
      </xs:sequence>
      <xs:attribute name='Context' type='xs:anyURI' use='optional'/>
      <xs:anyAttribute namespace='##other' processContents='lax'/>
    </xs:complexType>
    <xs:element name='RequestSecurityTokenCollection'
                type='wst:RequestSecurityTokenCollectionType'/>
    <xs:complexType name='RequestSecurityTokenCollectionType'>
      <xs:sequence>
        <xs:element name='RequestSecurityToken'
                    type='wst:AbstractRequestSecurityTokenType' minOccurs='2'
                    maxOccurs='unbounded'/>
      </xs:sequence>
    </xs:complexType>

    <xs:element name='RequestSecurityTokenResponseCollection'
                type='wst:RequestSecurityTokenResponseCollectionType'/>
    <xs:complexType name='RequestSecurityTokenResponseCollectionType'>
      <xs:sequence>
        <xs:element ref='wst:RequestSecurityTokenResponse' minOccurs='1'
                    maxOccurs='unbounded'/>
      </xs:sequence>
      <xs:anyAttribute namespace='##other' processContents='lax'/>
    </xs:complexType>

  </xs:schema>
</wsdl:types>

<!-- WS-Trust defines the following GEDs -->
<wsdl:message name="RequestSecurityTokenMsg">
  <wsdl:part name="request" element="wst:RequestSecurityToken"/>
</wsdl:message>
<wsdl:message name="RequestSecurityTokenResponseMsg">
  <wsdl:part name="response"
             element="wst:RequestSecurityTokenResponse"/>
</wsdl:message>
<wsdl:message name="RequestSecurityTokenCollectionMsg">
  <wsdl:part name="requestCollection"
             element="wst:RequestSecurityTokenCollection"/>
```

```
</wsdl:message>
<wsdl:message name="RequestSecurityTokenResponseCollectionMsg">
  <wsdl:part name="responseCollection"
             element="wst:RequestSecurityTokenResponseCollection"/>
</wsdl:message>

<!-- This portType an example of a Requestor (or other) endpoint that
Accepts SOAP-based challenges from a Security Token Service -->
<wsdl:portType name="WSSecurityRequestor">
  <wsdl:operation name="Challenge">
    <wsdl:input message="tns:RequestSecurityTokenResponseMsg"/>
    <wsdl:output message="tns:RequestSecurityTokenResponseMsg"/>
  </wsdl:operation>
</wsdl:portType>

<!-- This portType is an example of an STS supporting full protocol -->
<!--
    The wsdl:portType and data types are XML elements defined by the
    WS_Trust specification.  The wsdl:portType defines the endpoints
    supported in the STS implementation.  This WSDL defines all operations
    that an STS implementation can support.
-->
<wsdl:portType name="STS">
  <wsdl:operation name="Cancel">
    <wsdl:input
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Cancel"
      message="tns:RequestSecurityTokenMsg"/>
    <wsdl:output
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/CancelFinal"
      message="tns:RequestSecurityTokenResponseMsg"/>
  </wsdl:operation>
  <wsdl:operation name="Issue">
    <wsdl:input
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Issue"
      message="tns:RequestSecurityTokenMsg"/>
    <wsdl:output
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTRC/IssueFinal"
      message="tns:RequestSecurityTokenResponseCollectionMsg"/>
  </wsdl:operation>
  <wsdl:operation name="Renew">
    <wsdl:input
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Renew"
      message="tns:RequestSecurityTokenMsg"/>
    <wsdl:output
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/RenewFinal"
      message="tns:RequestSecurityTokenResponseMsg"/>
  </wsdl:operation>
  <wsdl:operation name="Validate">
    <wsdl:input
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Validate"
      message="tns:RequestSecurityTokenMsg"/>
    <wsdl:output
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/ValidateFinal"
      message="tns:RequestSecurityTokenResponseMsg"/>
  </wsdl:operation>
  <wsdl:operation name="KeyExchangeToken">
    <wsdl:input
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/KET"
```

```
      message="tns:RequestSecurityTokenMsg"/>
    <wsdl:output
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/KETFinal"
      message="tns:RequestSecurityTokenResponseMsg"/>
  </wsdl:operation>
  <wsdl:operation name="RequestCollection">
    <wsdl:input message="tns:RequestSecurityTokenCollectionMsg"/>
    <wsdl:output message="tns:RequestSecurityTokenResponseCollectionMsg"/>
  </wsdl:operation>
</wsdl:portType>

<!-- This portType is an example of an endpoint that accepts
Unsolicited RequestSecurityTokenResponse messages -->
<wsdl:portType name="SecurityTokenResponseService">
  <wsdl:operation name="RequestSecurityTokenResponse">
    <wsdl:input message="tns:RequestSecurityTokenResponseMsg"/>
  </wsdl:operation>
</wsdl:portType>

<!--
    The wsp:PolicyReference binds the security requirments on all the STS endpoints.
    The wsp:Policy wsu:Id="UT_policy" element is later in this file.
-->
<wsdl:binding name="UT_Binding" type="wstrust:STS">
  <wsp:PolicyReference URI="#UT_policy"/>
  <soap:binding style="document"
                transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="Issue">
    <soap:operation
      soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Issue"/>
    <wsdl:input>
      <wsp:PolicyReference
        URI="#Input_policy"/>
      <soap:body use="literal"/>
    </wsdl:input>
    <wsdl:output>
      <wsp:PolicyReference
        URI="#Output_policy"/>
      <soap:body use="literal"/>
    </wsdl:output>
  </wsdl:operation>
  <wsdl:operation name="Validate">
    <soap:operation
      soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Validate"/>
    <wsdl:input>
      <wsp:PolicyReference
        URI="#Input_policy"/>
      <soap:body use="literal"/>
    </wsdl:input>
    <wsdl:output>
      <wsp:PolicyReference
        URI="#Output_policy"/>
      <soap:body use="literal"/>
    </wsdl:output>
  </wsdl:operation>
  <wsdl:operation name="Cancel">
    <soap:operation
      soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Cancel"/>
```

```
      <wsdl:input>
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="Renew">
      <soap:operation
        soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Renew"/>
      <wsdl:input>
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="KeyExchangeToken">
      <soap:operation
        soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/KeyExchangeToken"/>
      <wsdl:input>
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="RequestCollection">
      <soap:operation
        soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/RequestCollection"/>
      <wsdl:input>
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>

  <wsdl:service name="SecurityTokenService">
    <wsdl:port name="UT_Port" binding="tns:UT_Binding">
      <soap:address location="http://localhost:8080/SecurityTokenService/UT"/>
    </wsdl:port>
  </wsdl:service>


  <wsp:Policy wsu:Id="UT_policy">
    <wsp:ExactlyOne>
      <wsp:All>
        <!--
            The sp:UsingAddressing element, indicates that the endpoints of this
            web service conforms to the WS-Addressing specification.  More detail
            can be found here: [http://www.w3.org/TR/2006/CR-ws-addr-wsdl-20060529]
        -->
        <wsap10:UsingAddressing/>
        <!--
            The sp:SymmetricBinding element indicates that security is provided
            at the SOAP layer and any initiator must authenticate itself by providing
```

```
            WSS UsernameToken credentials.
        -->
        <sp:SymmetricBinding
          xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
          <wsp:Policy>
            <!--
                In a symmetric binding, the keys used for encrypting and signing in both
                directions are derived from a single key, the one specified by the
                sp:ProtectionToken element.  The sp:X509Token sub-element declares this
                key to be a X.509 certificate and the

IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/Never"
                attribute adds the requirement that the token MUST NOT be included in
                any messages sent between the initiator and the recipient; rather, an
                external reference to the token should be used.  Lastly the WssX509V3Token10
                sub-element declares that the Username token presented by the initiator
                should be compliant with Web Services Security UsernameToken Profile
                1.0 specification. [
http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0.pdf ]
            -->
            <sp:ProtectionToken>
              <wsp:Policy>
                <sp:X509Token

sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/Never">
                  <wsp:Policy>
                    <sp:RequireDerivedKeys/>
                    <sp:RequireThumbprintReference/>
                    <sp:WssX509V3Token10/>
                  </wsp:Policy>
                </sp:X509Token>
              </wsp:Policy>
            </sp:ProtectionToken>
            <!--
                The sp:AlgorithmSuite element, requires the Basic256 algorithm suite
                be used in performing cryptographic operations.
            -->
            <sp:AlgorithmSuite>
              <wsp:Policy>
                <sp:Basic256/>
              </wsp:Policy>
            </sp:AlgorithmSuite>
            <!--
                The sp:Layout element,  indicates the layout rules to apply when adding
                items to the security header.  The sp:Lax sub-element indicates items
                are added to the security header in any order that conforms to
                WSS: SOAP Message Security.
            -->
            <sp:Layout>
              <wsp:Policy>
                <sp:Lax/>
              </wsp:Policy>
            </sp:Layout>
            <sp:IncludeTimestamp/>
            <sp:EncryptSignature/>
            <sp:OnlySignEntireHeadersAndBody/>
          </wsp:Policy>
        </sp:SymmetricBinding>
```

```
        <!--
            The sp:SignedSupportingTokens element declares that the security header
            of messages must contain a sp:UsernameToken and the token must be signed.
            The attribute
IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/AlwaysToRecip
on sp:UsernameToken indicates that the token MUST be included in all
            messages sent from initiator to the recipient and that the token MUST
            NOT be included in messages sent from the recipient to the initiator.
            And finally the element sp:WssUsernameToken10 is a policy assertion
            indicating the Username token should be as defined in  Web Services
            Security UsernameToken Profile 1.0
        -->
        <sp:SignedSupportingTokens
          xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
          <wsp:Policy>
            <sp:UsernameToken

sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/AlwaysToRe
<wsp:Policy>
                <sp:WssUsernameToken10/>
              </wsp:Policy>
            </sp:UsernameToken>
          </wsp:Policy>
        </sp:SignedSupportingTokens>
        <!--
            The sp:Wss11 element declares WSS: SOAP Message Security 1.1 options
            to be supported by the STS.  These particular elements generally refer
            to how keys are referenced within the SOAP envelope.  These are normally
            handled by CXF.
        -->
        <sp:Wss11
          xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
          <wsp:Policy>
            <sp:MustSupportRefKeyIdentifier/>
            <sp:MustSupportRefIssuerSerial/>
            <sp:MustSupportRefThumbprint/>
            <sp:MustSupportRefEncryptedKey/>
          </wsp:Policy>
        </sp:Wss11>
        <!--
            The sp:Trust13 element declares controls for WS-Trust 1.3 options.
            They are policy assertions related to exchanges specifically with
            client and server challenges and entropy behaviors.  Again these are
            normally handled by CXF.
        -->
        <sp:Trust13
          xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
          <wsp:Policy>
            <sp:MustSupportIssuedTokens/>
            <sp:RequireClientEntropy/>
            <sp:RequireServerEntropy/>
          </wsp:Policy>
        </sp:Trust13>
      </wsp:All>
    </wsp:ExactlyOne>
  </wsp:Policy>
```

```
  <wsp:Policy wsu:Id="Input_policy">
    <wsp:ExactlyOne>
      <wsp:All>
        <sp:SignedParts
          xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
          <sp:Body/>
          <sp:Header Name="To"
                     Namespace="http://www.w3.org/2005/08/addressing"/>
          <sp:Header Name="From"
                     Namespace="http://www.w3.org/2005/08/addressing"/>
          <sp:Header Name="FaultTo"
                     Namespace="http://www.w3.org/2005/08/addressing"/>
          <sp:Header Name="ReplyTo"
                     Namespace="http://www.w3.org/2005/08/addressing"/>
          <sp:Header Name="MessageID"
                     Namespace="http://www.w3.org/2005/08/addressing"/>
          <sp:Header Name="RelatesTo"
                     Namespace="http://www.w3.org/2005/08/addressing"/>
          <sp:Header Name="Action"
                     Namespace="http://www.w3.org/2005/08/addressing"/>
        </sp:SignedParts>
      </wsp:All>
    </wsp:ExactlyOne>
  </wsp:Policy>

  <wsp:Policy wsu:Id="Output_policy">
    <wsp:ExactlyOne>
      <wsp:All>
        <sp:SignedParts
          xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
          <sp:Body/>
          <sp:Header Name="To"
                     Namespace="http://www.w3.org/2005/08/addressing"/>
          <sp:Header Name="From"
                     Namespace="http://www.w3.org/2005/08/addressing"/>
          <sp:Header Name="FaultTo"
                     Namespace="http://www.w3.org/2005/08/addressing"/>
          <sp:Header Name="ReplyTo"
                     Namespace="http://www.w3.org/2005/08/addressing"/>
          <sp:Header Name="MessageID"
                     Namespace="http://www.w3.org/2005/08/addressing"/>
          <sp:Header Name="RelatesTo"
                     Namespace="http://www.w3.org/2005/08/addressing"/>
          <sp:Header Name="Action"
                     Namespace="http://www.w3.org/2005/08/addressing"/>
        </sp:SignedParts>
      </wsp:All>
    </wsp:ExactlyOne>
  </wsp:Policy>

</wsdl:definitions>
```

STS's implementation class

The Apache CXF's STS, SecurityTokenServiceProvider, is a web service provider that is compliant with the protocols and functionality defined by the WS-Trust specification. It has a modular architecture. Many of its components are configurable or replaceable and there are many optional features that are enabled by implementing and configuring plug-ins. Users can customize their own STS by extending from SecurityTokenServiceProvider and overriding the default settings. Extensive information about the CXF's STS configurable and pluggable components can be found here.

This STS implementation class, SampleSTSBearer, is a POJO that extends from SecurityTokenServiceProvider. Note that the class is defined with a WebServiceProvider annotation and not a WebService annotation. This annotation defines the service as a Provider-based endpoint, meaning it supports a more messaging-oriented approach to Web services. In particular, it signals that the exchanged messages will be XML documents of some type. SecurityTokenServiceProvider is an implementation of the javax.xml.ws.Provider interface. In comparison the WebService annotation defines a (service endpoint interface) SEI-based endpoint which supports message exchange via SOAP envelopes.

As was done in the BearerImpl class, the WSS4J annotations EndpointProperties and EndpointProperty are providing endpoint configuration for the CXF runtime. The first EndpointProperty statement in the listing is declaring the user's name to use for the message signature. It is used as the alias name in the keystore to get the user's cert and private key for signature. The next two EndpointProperty statements declares the Java properties file that contains the (Merlin) crypto configuration information. In this case both for signing and encrypting the messages. WSS4J reads this file and extra required information for message handling. The last EndpointProperty statement declares the STSBearerCallbackHandler implementation class. It is used to obtain the user's password for the certificates in the keystore file.

In this implementation we are customizing the operations of token issuance, token validation and their static properties.

StaticSTSProperties is used to set select properties for configuring resources in the STS. You may think this is a duplication of the settings made with the WSS4J annotations. The values are the same but the underlaying structures being set are different, thus this information must be declared in both places.

The setIssuer setting is important because it uniquely identifies the issuing STS. The issuer string is embedded in issued tokens and, when validating tokens, the STS checks the issuer string value. Consequently, it is important to use the issuer string in a consistent way, so that the STS can recognize the tokens that it has issued.

The setEndpoints call allows the declaration of a set of allowed token recipients by address. The addresses are specified as reg-ex patterns.

TokenIssueOperation has a modular structure. This allows custom behaviors to be injected into the processing of messages. In this case we are overriding the SecurityTokenServiceProvider's default behavior and performing SAML token processing. CXF provides an implementation of a SAMLTokenProvider which we are using rather than writing our own.

Learn more about the SAMLTokenProvider here.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.stsbearer;

import org.apache.cxf.annotations.EndpointProperties;
```

```
import org.apache.cxf.annotations.EndpointProperty;
import org.apache.cxf.sts.StaticSTSProperties;
import org.apache.cxf.sts.operation.TokenIssueOperation;
import org.apache.cxf.sts.service.ServiceMBean;
import org.apache.cxf.sts.service.StaticService;
import org.apache.cxf.sts.token.provider.SAMLTokenProvider;
import org.apache.cxf.ws.security.sts.provider.SecurityTokenServiceProvider;

import javax.xml.ws.WebServiceProvider;
import java.util.Arrays;
import java.util.LinkedList;
import java.util.List;

@WebServiceProvider(serviceName = "SecurityTokenService",
      portName = "UT_Port",
      targetNamespace = "http://docs.oasis-open.org/ws-sx/ws-trust/200512/",
      wsdlLocation = "WEB-INF/wsdl/bearer-ws-trust-1.4-service.wsdl")
//be sure to have dependency on org.apache.cxf module when on AS7, otherwise Apache CXF
annotations are ignored
@EndpointProperties(value = {
      @EndpointProperty(key = "ws-security.signature.username", value = "mystskey"),
      @EndpointProperty(key = "ws-security.signature.properties", value =
"stsKeystore.properties"),
      @EndpointProperty(key = "ws-security.callback-handler", value =
"org.jboss.test.ws.jaxws.samples.wsse.policy.trust.stsbearer.STSBearerCallbackHandler")
})
public class SampleSTSBearer extends SecurityTokenServiceProvider
{

   public SampleSTSBearer() throws Exception
   {
      super();

      StaticSTSProperties props = new StaticSTSProperties();
      props.setSignatureCryptoProperties("stsKeystore.properties");
      props.setSignatureUsername("mystskey");
      props.setCallbackHandlerClass(STSBearerCallbackHandler.class.getName());
      props.setEncryptionCryptoProperties("stsKeystore.properties");
      props.setEncryptionUsername("myservicekey");
      props.setIssuer("DoubleItSTSIssuer");

      List<ServiceMBean> services = new LinkedList<ServiceMBean>();
      StaticService service = new StaticService();
      service.setEndpoints(Arrays.asList(
         "https://localhost:(\\d)*/jaxws-samples-wsse-policy-trust-bearer/BearerService",
         "https://\\[::1\\]:(\\d)*/jaxws-samples-wsse-policy-trust-bearer/BearerService",

"https://\\[0:0:0:0:0:0:0:1\\]:(\\d)*/jaxws-samples-wsse-policy-trust-bearer/BearerService"
      ));
      services.add(service);

      TokenIssueOperation issueOperation = new TokenIssueOperation();
      issueOperation.getTokenProviders().add(new SAMLTokenProvider());
      issueOperation.setServices(services);
      issueOperation.setStsProperties(props);
      this.setIssueOperation(issueOperation);
   }
}
```

STSBearerCallbackHandler

STSBearerCallbackHandler is a callback handler for the WSS4J Crypto API. It is used to obtain the password for the private key in the keystore. This class enables CXF to retrieve the password of the user name to use for the message signature.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.stsbearer;

import org.jboss.wsf.stack.cxf.extensions.security.PasswordCallbackHandler;

import java.util.HashMap;
import java.util.Map;

public class STSBearerCallbackHandler extends PasswordCallbackHandler
{
   public STSBearerCallbackHandler()
   {
      super(getInitMap());
   }

   private static Map<String, String> getInitMap()
   {
      Map<String, String> passwords = new HashMap<String, String>();
      passwords.put("mystskey", "stskpass");
      passwords.put("alice", "clarinet");
      return passwords;
   }
}
```

Crypto properties and keystore files

WSS4J's Crypto implementation is loaded and configured via a Java properties file that contains Crypto configuration data. The file contains implementation-specific properties such as a keystore location, password, default alias and the like. This application is using the Merlin implementation. File stsKeystore.properties contains this information.

File servicestore.jks, is a Java KeyStore (JKS) repository. It contains self signed certificates for myservicekey and mystskey. *Self signed certificates are not appropriate for production use.*

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=stsspass
org.apache.ws.security.crypto.merlin.keystore.file=stsstore.jks
```

MANIFEST.MF

When deployed on WildFly, this application requires access to the JBossWs and CXF APIs provided in modules org.jboss.ws.cxf.jbossws-cxf-client and org.apache.cxf. The Apache CXF internals, org.apache.cxf.impl, are needed to build the STS configuration in the `SampleSTS` constructor. The dependency statement directs the server to provide them at deployment.

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.8.2
Created-By: 1.7.0_25-b15 (Oracle Corporation)
Dependencies: org.jboss.ws.cxf.jbossws-cxf-client,org.apache.cxf.impl
```

Web service requester

This section examines the crucial elements in calling a web service that implements endpoint security as described in the SAML Bearer scenario. The components that will be discussed are.

- Web service requester's implementation
- ClientCallbackHandler
- Crypto properties and keystore files

Web service requester Implementation

The ws-requester, the client, uses standard procedures for creating a reference to the web service. To address the endpoint security requirements, the web service's "Request Context" is configured with the information needed in message generation. In addition, the STSClient that communicates with the STS is configured with similar values. Note the key strings ending with a ".it" suffix. This suffix flags these settings as belonging to the STSClient. The internal CXF code assigns this information to the STSClient that is auto-generated for this service call.

There is an alternate method of setting up the STSCLient. The user may provide their own instance of the STSClient. The CXF code will use this object and not auto-generate one. When providing the STSClient in this way, the user must provide a org.apache.cxf.Bus for it and the configuration keys must not have the ".it" suffix. This is used in the ActAs and OnBehalfOf examples.

```
String serviceURL = "https://" + getServerHost() +
":8443/jaxws-samples-wsse-policy-trust-bearer/BearerService";

  final QName serviceName = new
QName("http://www.jboss.org/jbossws/ws-extensions/bearerwssecuritypolicy", "BearerService");
  Service service = Service.create(new URL(serviceURL + "?wsdl"), serviceName);
  BearerIface proxy = (BearerIface) service.getPort(BearerIface.class);

  Map<String, Object> ctx = ((BindingProvider)proxy).getRequestContext();

  // set the security related configuration information for the service "request"
  ctx.put(SecurityConstants.CALLBACK_HANDLER, new ClientCallbackHandler());
  ctx.put(SecurityConstants.SIGNATURE_PROPERTIES,
    Thread.currentThread().getContextClassLoader().getResource(
    "META-INF/clientKeystore.properties"));
  ctx.put(SecurityConstants.ENCRYPT_PROPERTIES,
    Thread.currentThread().getContextClassLoader().getResource(
    "META-INF/clientKeystore.properties"));
  ctx.put(SecurityConstants.SIGNATURE_USERNAME, "myclientkey");
  ctx.put(SecurityConstants.ENCRYPT_USERNAME, "myservicekey");

  //-- Configuration settings that will be transfered to the STSClient
  // "alice" is the name provided for the WSS Username. Her password will
  // be retreived from the ClientCallbackHander by the STSClient.
  ctx.put(SecurityConstants.USERNAME + ".it", "alice");
  ctx.put(SecurityConstants.CALLBACK_HANDLER + ".it", new ClientCallbackHandler());
  ctx.put(SecurityConstants.ENCRYPT_PROPERTIES + ".it",
    Thread.currentThread().getContextClassLoader().getResource(
    "META-INF/clientKeystore.properties"));
  ctx.put(SecurityConstants.ENCRYPT_USERNAME + ".it", "mystskey");
  ctx.put(SecurityConstants.STS_TOKEN_USERNAME + ".it", "myclientkey");
  ctx.put(SecurityConstants.STS_TOKEN_PROPERTIES + ".it",
    Thread.currentThread().getContextClassLoader().getResource(
    "META-INF/clientKeystore.properties"));
  ctx.put(SecurityConstants.STS_TOKEN_USE_CERT_FOR_KEYINFO + ".it", "true");

  proxy.sayHello();
```

ClientCallbackHandler
https://docs.jboss.org/author/display/JBWS/WS-Trust+and+STS#WS-TrustandSTS-ClientCallbackHandler

ClientCallbackHandler is a callback handler for the WSS4J Crypto API. It is used to obtain the password for the private key in the keystore. This class enables CXF to retrieve the password of the user name to use for the message signature. Note that "alice" and her password have been provided here. This information is not in the (JKS) keystore but provided in the WildFly security domain. It was declared in file jbossws-users.properties.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.shared;

import java.io.IOException;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import org.apache.ws.security.WSPasswordCallback;

public class ClientCallbackHandler implements CallbackHandler {

    public void handle(Callback[] callbacks) throws IOException,
            UnsupportedCallbackException {
        for (int i = 0; i < callbacks.length; i++) {
            if (callbacks[i] instanceof WSPasswordCallback) {
                WSPasswordCallback pc = (WSPasswordCallback) callbacks[i];
                if ("myclientkey".equals(pc.getIdentifier())) {
                    pc.setPassword("ckpass");
                    break;
                } else if ("alice".equals(pc.getIdentifier())) {
                    pc.setPassword("clarinet");
                    break;
                } else if ("bob".equals(pc.getIdentifier())) {
                    pc.setPassword("trombone");
                    break;
                } else if ("myservicekey".equals(pc.getIdentifier())) {  // rls test  added for
bearer test
                   pc.setPassword("skpass");
                   break;
                }
            }
        }
    }
}
```

Crypto properties and keystore files
https://docs.jboss.org/author/display/JBWS/WS-Trust+and+STS#WS-TrustandSTS-RequesterCryptopropertie

WSS4J's Crypto implementation is loaded and configured via a Java properties file that contains Crypto configuration data. The file contains implementation-specific properties such as a keystore location, password, default alias and the like. This application is using the Merlin implementation. File clientKeystore.properties contains this information.

File clientstore.jks, is a Java KeyStore (JKS) repository. It contains self signed certificates for myservicekey and mystskey. *Self signed certificates are not appropriate for production use.*

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=cspass
org.apache.ws.security.crypto.merlin.keystore.alias=myclientkey
org.apache.ws.security.crypto.merlin.keystore.file=META-INF/clientstore.jks
```

SAML Holder-Of-Key Assertion Scenario

SAML Holder-Of-Key Assertion Scenario

WS-Trust deals with managing software security tokens. A SAML assertion is a type of security token. In the Holder-Of-Key method, the STS creates a SAML token containing the client's public key and signs the SAML token with its private key. The client includes the SAML token and signs the outgoing soap envelope to the web service with its private key. The web service validates the SOAP message and the SAML token.

Implementation of this scenario has the following requirements.

- SAML tokens with a Holder-Of-Key subject confirmation method must be protected so the token can not be snooped. In most cases, a Holder-Of-Key token combined with HTTPS is sufficient to prevent "a man in the middle" getting possession of the token. This means a security policy that uses a sp:TransportBinding and sp:HttpsToken.
- A Holder-Of-Key token has no encryption or signing keys associated with it, therefore a sp:IssuedToken of SymmetricKey or PublicKey keyType should be used with a sp:SignedEndorsingSupportingTokens.

Web service Provider

This section examines the web service elements for the SAML Holder-Of-Key scenario. The components are

- Web service provider's WSDL
- SSL configuration
- Web service provider's Interface and Implementation classes.
- Crypto properties and keystore files
- MANIFEST.MF

Web service provider WSDL

The web service provider is a contract-first endpoint. All the WS-trust and security policies for it are declared in the WSDL, HolderOfKeyService.wsdl. For this scenario a ws-requester is required to present a SAML 2.0 token of SymmetricKey keyType, issued from a designed STS. The address of the STS is provided in the WSDL. A transport binding policy is used. The token is declared to be signed and endorsed, sp:SignedEndorsingSupportingTokens. A detailed explanation of the security settings are provided in the comments in the listing below.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions
targetNamespace="http://www.jboss.org/jbossws/ws-extensions/holderofkeywssecuritypolicy"
             name="HolderOfKeyService"
        xmlns:tns="http://www.jboss.org/jbossws/ws-extensions/holderofkeywssecuritypolicy"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns="http://schemas.xmlsoap.org/wsdl/"
        xmlns:wsp="http://www.w3.org/ns/ws-policy"
        xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"

xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
    xmlns:wsaws="http://www.w3.org/2005/08/addressing"
    xmlns:wsx="http://schemas.xmlsoap.org/ws/2004/09/mex"
    xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702"
    xmlns:t="http://docs.oasis-open.org/ws-sx/ws-trust/200512">

  <types>
    <xsd:schema>
      <xsd:import
namespace="http://www.jboss.org/jbossws/ws-extensions/holderofkeywssecuritypolicy"
                  schemaLocation="HolderOfKeyService_schema1.xsd"/>
    </xsd:schema>
```

```
  </types>
  <message name="sayHello">
    <part name="parameters" element="tns:sayHello"/>
  </message>
  <message name="sayHelloResponse">
    <part name="parameters" element="tns:sayHelloResponse"/>
  </message>
  <portType name="HolderOfKeyIface">
    <operation name="sayHello">
      <input message="tns:sayHello"/>
      <output message="tns:sayHelloResponse"/>
    </operation>
  </portType>
<!--
        The wsp:PolicyReference binds the security requirments on all the endpoints.
        The wsp:Policy wsu:Id="#TransportSAML2HolderOfKeyPolicy" element is defined later in
this file.
-->
  <binding name="HolderOfKeyServicePortBinding" type="tns:HolderOfKeyIface">
    <wsp:PolicyReference URI="#TransportSAML2HolderOfKeyPolicy" />
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
    <operation name="sayHello">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>
<!--
  The soap:address has been defined to use JBoss's https port, 8443.  This is
  set in conjunction with the sp:TransportBinding policy for https.
-->
  <service name="HolderOfKeyService">
    <port name="HolderOfKeyServicePort" binding="tns:HolderOfKeyServicePortBinding">
      <soap:address
location="https://@jboss.bind.address@:8443/jaxws-samples-wsse-policy-trust-holderofkey/HolderOfKe
</port>
  </service>


  <wsp:Policy wsu:Id="TransportSAML2HolderOfKeyPolicy">
    <wsp:ExactlyOne>
      <wsp:All>
  <!--
        The wsam:Addressing element, indicates that the endpoints of this
        web service MUST conform to the WS-Addressing specification.  The
        attribute wsp:Optional="false" enforces this assertion.
  -->
        <wsam:Addressing wsp:Optional="false">
          <wsp:Policy />
        </wsam:Addressing>
<!--
  The sp:TransportBinding element indicates that security is provided by the
  message exchange transport medium, https.  WS-Security policy specification
  defines the sp:HttpsToken for use in exchanging messages transmitted over HTTPS.
```

```
-->
          <sp:TransportBinding
            xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
            <wsp:Policy>
              <sp:TransportToken>
                <wsp:Policy>
                  <sp:HttpsToken>
                    <wsp:Policy/>
                  </sp:HttpsToken>
                </wsp:Policy>
              </sp:TransportToken>
<!--
     The sp:AlgorithmSuite element, requires the TripleDes algorithm suite
     be used in performing cryptographic operations.
-->
              <sp:AlgorithmSuite>
                <wsp:Policy>
                  <sp:TripleDes />
                </wsp:Policy>
              </sp:AlgorithmSuite>
<!--
     The sp:Layout element,  indicates the layout rules to apply when adding
     items to the security header.  The sp:Lax sub-element indicates items
     are added to the security header in any order that conforms to
     WSS: SOAP Message Security.
-->
              <sp:Layout>
                <wsp:Policy>
                  <sp:Lax />
                </wsp:Policy>
              </sp:Layout>
              <sp:IncludeTimestamp />
            </wsp:Policy>
          </sp:TransportBinding>

<!--
  The sp:SignedEndorsingSupportingTokens, when transport level security level is
  used there will be no message signature and the signature generated by the
  supporting token will sign the Timestamp.
-->
        <sp:SignedEndorsingSupportingTokens
          xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
          <wsp:Policy>
<!--
  The sp:IssuedToken element asserts that a SAML 2.0 security token of type
  Bearer is expected from the STS.  The

sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/AlwaysToRe
attribute instructs the runtime to include the initiator's public key
  with every message sent to the recipient.

  The sp:RequestSecurityTokenTemplate element directs that all of the
  children of this element will be copied directly into the body of the
  RequestSecurityToken (RST) message that is sent to the STS when the
  initiator asks the STS to issue a token.
-->
            <sp:IssuedToken
```

```
sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/AlwaysToRe
<sp:RequestSecurityTokenTemplate>

<t:TokenType>http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV2.0</t:TokenType
<!--
   KeyType of "SymmetricKey", the client must prove to the WS service that it
   possesses a particular symmetric session key.
 -->

<t:KeyType>http://docs.oasis-open.org/ws-sx/ws-trust/200512/SymmetricKey</t:KeyType>
              </sp:RequestSecurityTokenTemplate>
              <wsp:Policy>
                <sp:RequireInternalReference />
              </wsp:Policy>
<!--
  The sp:Issuer element defines the STS's address and endpoint information
  This information is used by the STSClient.
-->
              <sp:Issuer>

<wsaws:Address>http://@jboss.bind.address@:8080/jaxws-samples-wsse-policy-trust-sts-holderofkey/Se
<wsaws:Metadata
                  xmlns:wsdli="http://www.w3.org/2006/01/wsdl-instance"

wsdli:wsdlLocation="http://@jboss.bind.address@:8080/jaxws-samples-wsse-policy-trust-sts-holderofk
<wsaw:ServiceName
                    xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl"
                    xmlns:stsns="http://docs.oasis-open.org/ws-sx/ws-trust/200512/"
                    EndpointName="UT_Port">stsns:SecurityTokenService</wsaw:ServiceName>
                </wsaws:Metadata>
              </sp:Issuer>

            </sp:IssuedToken>
          </wsp:Policy>
        </sp:SignedEndorsingSupportingTokens>
<!--
    The sp:Wss11 element declares WSS: SOAP Message Security 1.1 options
    to be supported by the STS.  These particular elements generally refer
    to how keys are referenced within the SOAP envelope.  These are normally
    handled by CXF.
-->
        <sp:Wss11>
          <wsp:Policy>
            <sp:MustSupportRefIssuerSerial />
            <sp:MustSupportRefThumbprint />
            <sp:MustSupportRefEncryptedKey />
          </wsp:Policy>
        </sp:Wss11>
<!--
    The sp:Trust13 element declares controls for WS-Trust 1.3 options.
    They are policy assertions related to exchanges specifically with
    client and server challenges and entropy behaviors.  Again these are
    normally handled by CXF.
-->
        <sp:Trust13>
          <wsp:Policy>
            <sp:MustSupportIssuedTokens />
            <sp:RequireClientEntropy />
```

```
            <sp:RequireServerEntropy />
          </wsp:Policy>
        </sp:Trust13>
      </wsp:All>
    </wsp:ExactlyOne>
  </wsp:Policy>

</definitions>
```

SSL configuration
https://docs.jboss.org/author/display/JBWS/WS-Trust+and+STS#WS-TrustandSTS-SSLconfiguration

This web service is using https, therefore the JBoss server must be configured to provide SSL support in the Web subsystem. There are 2 components to SSL configuration.

- create a certificate keystore
- declare an SSL connector in the Web subsystem of the JBoss server configuration file.

Follow the directions in the, "*Using the pure Java implementation supplied by JSSE*" section in the [SSL Setup Guide|../../../../../../../../../../display/WFLY8/SSL+setup+guide|||\||].

Here is an example of an SSL connector declaration.

```
<subsystem xmlns="urn:jboss:domain:web:1.4" default-virtual-server="default-host"
native="false">
.....
  <connector name="jbws-https-connector" protocol="HTTP/1.1" scheme="https"
socket-binding="https" secure="true" enabled="true">
    <ssl key-alias="tomcat" password="changeit"
certificate-key-file="/myJbossHome/security/test.keystore" verify-client="false"/>
  </connector>
...
```

Web service Interface
The web service provider interface class, HolderOfKeyIface, is a simple straight forward web service definition.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.holderofkey;

import javax.jws.WebMethod;
import javax.jws.WebService;

@WebService
(
   targetNamespace = "http://www.jboss.org/jbossws/ws-extensions/holderofkeywssecuritypolicy"
)
public interface HolderOfKeyIface {
   @WebMethod
   String sayHello();
}
```

Web service Implementation

The web service provider implementation class, HolderOfKeyImpl, is a simple POJO. It uses the standard WebService annotation to define the service endpoint. In addition there are two Apache CXF annotations, EndpointProperties and EndpointProperty used for configuring the endpoint for the CXF runtime. These annotations come from the Apache WSS4J project, which provides a Java implementation of the primary WS-Security standards for Web Services. These annotations are programmatically adding properties to the endpoint. With plain Apache CXF, these properties are often set via the <jaxws:properties> element on the <jaxws:endpoint> element in the Spring config; these annotations allow the properties to be configured in the code.

WSS4J uses the Crypto interface to get keys and certificates for signature creation/verification, as is asserted by the WSDL for this service. The WSS4J configuration information being provided by HolderOfKeyImpl is for Crypto's Merlin implementation. More information will be provided about this in the keystore section.

The first EndpointProperty statement in the listing disables ensurance of compliance with the Basic Security Profile 1.1. The next EndpointProperty statements declares the Java properties file that contains the (Merlin) crypto configuration information. The last EndpointProperty statement declares the STSHolderOfKeyCallbackHandler implementation class. It is used to obtain the user's password for the certificates in the keystore file.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.holderofkey;

import org.apache.cxf.annotations.EndpointProperties;
import org.apache.cxf.annotations.EndpointProperty;

import javax.jws.WebService;

@WebService
   (
      portName = "HolderOfKeyServicePort",
      serviceName = "HolderOfKeyService",
      wsdlLocation = "WEB-INF/wsdl/HolderOfKeyService.wsdl",
      targetNamespace =
"http://www.jboss.org/jbossws/ws-extensions/holderofkeywssecuritypolicy",
      endpointInterface =
"org.jboss.test.ws.jaxws.samples.wsse.policy.trust.holderofkey.HolderOfKeyIface"
   )
@EndpointProperties(value = {
   @EndpointProperty(key = "ws-security.is-bsp-compliant", value = "false"),
   @EndpointProperty(key = "ws-security.signature.properties", value =
"serviceKeystore.properties"),
   @EndpointProperty(key = "ws-security.callback-handler", value =
"org.jboss.test.ws.jaxws.samples.wsse.policy.trust.holderofkey.HolderOfKeyCallbackHandler")
})
public class HolderOfKeyImpl implements HolderOfKeyIface
{
   public String sayHello()
   {
      return "Holder-Of-Key WS-Trust Hello World!";
   }
}
```

Crypto properties and keystore files

WSS4J's Crypto implementation is loaded and configured via a Java properties file that contains Crypto configuration data. The file contains implementation-specific properties such as a keystore location, password, default alias and the like. This application is using the Merlin implementation. File serviceKeystore.properties contains this information.

File servicestore.jks, is a Java KeyStore (JKS) repository. It contains self signed certificates for myservicekey and mystskey. *Self signed certificates are not appropriate for production use.*

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=sspass
org.apache.ws.security.crypto.merlin.keystore.alias=myservicekey
org.apache.ws.security.crypto.merlin.keystore.file=servicestore.jks
```

MANIFEST.MF

https://docs.jboss.org/author/display/JBWS/WS-Trust+and+STS#WS-TrustandSTS-MANIFEST.MF

When deployed on WildFly this application requires access to the JBossWs and CXF APIs provided in module org.jboss.ws.cxf.jbossws-cxf-client. The dependency statement directs the server to provide them at deployment.

```
Manifest-Version:1.0
Ant-Version: Apache Ant1.8.2
Created-By:1.7.0_25-b15 (Oracle Corporation)
Dependencies: org.jboss.ws.cxf.jbossws-cxf-client
```

Security Token Service

This section examines the crucial elements in providing the Security Token Service functionality for providing a SAML Holder-Of-Key token. The components that will be discussed are.

- Security Domain
- STS's WSDL
- STS's implementation class
- STSBearerCallbackHandler
- Crypto properties and keystore files
- MANIFEST.MF

Security Domain

The STS requires a JBoss security domain be configured. The jboss-web.xml descriptor declares a named security domain,"JBossWS-trust-sts" to be used by this service for authentication. This security domain requires two properties files and the addition of a security-domain declaration in the JBoss server configuration file.

For this scenario the domain needs to contain user *alice*, password *clarinet*, and role *friend*. See the listings below for jbossws-users.properties and jbossws-roles.properties. In addition the following XML must be added to the JBoss security subsystem in the server configuration file. Replace "**SOME_PATH**" with appropriate information.

```
<security-domain name="JBossWS-trust-sts">
  <authentication>
   <login-module code="UsersRoles" flag="required">
     <module-option name="usersProperties" value="/SOME_PATH/jbossws-users.properties"/>
     <module-option name="unauthenticatedIdentity" value="anonymous"/>
     <module-option name="rolesProperties" value="/SOME_PATH/jbossws-roles.properties"/>
   </login-module>
  </authentication>
</security-domain>
```

jboss-web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jboss-web PUBLIC"-//JBoss//DTD Web Application 2.4//EN" ">
<jboss-web>
  <security-domain>java:/jaas/JBossWS-trust-sts</security-domain>
</jboss-web>
```

jbossws-users.properties

```
# A sample users.properties filefor use with the UsersRolesLoginModule
alice=clarinet
```

jbossws-roles.properties

```
# A sample roles.properties filefor use with the UsersRolesLoginModule
alice=friend
```

STS's WSDL

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
  targetNamespace="http://docs.oasis-open.org/ws-sx/ws-trust/200512/"
```

```
xmlns:tns="http://docs.oasis-open.org/ws-sx/ws-trust/200512/"
xmlns:wstrust="http://docs.oasis-open.org/ws-sx/ws-trust/200512/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:wsap10="http://www.w3.org/2006/05/addressing/wsdl"
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
xmlns:wsp="http://www.w3.org/ns/ws-policy"
xmlns:wst="http://docs.oasis-open.org/ws-sx/ws-trust/200512"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata">

<wsdl:types>
  <xs:schema elementFormDefault="qualified"
             targetNamespace='http://docs.oasis-open.org/ws-sx/ws-trust/200512'>

    <xs:element name='RequestSecurityToken'
                type='wst:AbstractRequestSecurityTokenType'/>
    <xs:element name='RequestSecurityTokenResponse'
                type='wst:AbstractRequestSecurityTokenType'/>

    <xs:complexType name='AbstractRequestSecurityTokenType'>
      <xs:sequence>
        <xs:any namespace='##any' processContents='lax' minOccurs='0'
                maxOccurs='unbounded'/>
      </xs:sequence>
      <xs:attribute name='Context' type='xs:anyURI' use='optional'/>
      <xs:anyAttribute namespace='##other' processContents='lax'/>
    </xs:complexType>
    <xs:element name='RequestSecurityTokenCollection'
                type='wst:RequestSecurityTokenCollectionType'/>
    <xs:complexType name='RequestSecurityTokenCollectionType'>
      <xs:sequence>
        <xs:element name='RequestSecurityToken'
                    type='wst:AbstractRequestSecurityTokenType' minOccurs='2'
                    maxOccurs='unbounded'/>
      </xs:sequence>
    </xs:complexType>

    <xs:element name='RequestSecurityTokenResponseCollection'
                type='wst:RequestSecurityTokenResponseCollectionType'/>
    <xs:complexType name='RequestSecurityTokenResponseCollectionType'>
      <xs:sequence>
        <xs:element ref='wst:RequestSecurityTokenResponse' minOccurs='1'
                    maxOccurs='unbounded'/>
      </xs:sequence>
      <xs:anyAttribute namespace='##other' processContents='lax'/>
    </xs:complexType>

  </xs:schema>
</wsdl:types>

<!-- WS-Trust defines the following GEDs -->
<wsdl:message name="RequestSecurityTokenMsg">
  <wsdl:part name="request" element="wst:RequestSecurityToken"/>
</wsdl:message>
<wsdl:message name="RequestSecurityTokenResponseMsg">
  <wsdl:part name="response"
             element="wst:RequestSecurityTokenResponse"/>
```

```
</wsdl:message>
<wsdl:message name="RequestSecurityTokenCollectionMsg">
  <wsdl:part name="requestCollection"
             element="wst:RequestSecurityTokenCollection"/>
</wsdl:message>
<wsdl:message name="RequestSecurityTokenResponseCollectionMsg">
  <wsdl:part name="responseCollection"
             element="wst:RequestSecurityTokenResponseCollection"/>
</wsdl:message>

<!-- This portType an example of a Requestor (or other) endpoint that
       Accepts SOAP-based challenges from a Security Token Service -->
<wsdl:portType name="WSSecurityRequestor">
  <wsdl:operation name="Challenge">
    <wsdl:input message="tns:RequestSecurityTokenResponseMsg"/>
    <wsdl:output message="tns:RequestSecurityTokenResponseMsg"/>
  </wsdl:operation>
</wsdl:portType>

<!-- This portType is an example of an STS supporting full protocol -->
<wsdl:portType name="STS">
  <wsdl:operation name="Cancel">
    <wsdl:input
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Cancel"
      message="tns:RequestSecurityTokenMsg"/>
    <wsdl:output
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/CancelFinal"
      message="tns:RequestSecurityTokenResponseMsg"/>
  </wsdl:operation>
  <wsdl:operation name="Issue">
    <wsdl:input
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Issue"
      message="tns:RequestSecurityTokenMsg"/>
    <wsdl:output
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTRC/IssueFinal"
      message="tns:RequestSecurityTokenResponseCollectionMsg"/>
  </wsdl:operation>
  <wsdl:operation name="Renew">
    <wsdl:input
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Renew"
      message="tns:RequestSecurityTokenMsg"/>
    <wsdl:output
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/RenewFinal"
      message="tns:RequestSecurityTokenResponseMsg"/>
  </wsdl:operation>
  <wsdl:operation name="Validate">
    <wsdl:input
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Validate"
      message="tns:RequestSecurityTokenMsg"/>
    <wsdl:output
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/ValidateFinal"
      message="tns:RequestSecurityTokenResponseMsg"/>
  </wsdl:operation>
  <wsdl:operation name="KeyExchangeToken">
    <wsdl:input
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/KET"
      message="tns:RequestSecurityTokenMsg"/>
    <wsdl:output
```

```
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/KETFinal"
      message="tns:RequestSecurityTokenResponseMsg"/>
  </wsdl:operation>
  <wsdl:operation name="RequestCollection">
    <wsdl:input message="tns:RequestSecurityTokenCollectionMsg"/>
    <wsdl:output message="tns:RequestSecurityTokenResponseCollectionMsg"/>
  </wsdl:operation>
</wsdl:portType>

<!-- This portType is an example of an endpoint that accepts
       Unsolicited RequestSecurityTokenResponse messages -->
<wsdl:portType name="SecurityTokenResponseService">
  <wsdl:operation name="RequestSecurityTokenResponse">
    <wsdl:input message="tns:RequestSecurityTokenResponseMsg"/>
  </wsdl:operation>
</wsdl:portType>

<wsdl:binding name="UT_Binding" type="wstrust:STS">
  <wsp:PolicyReference URI="#UT_policy"/>
  <soap:binding style="document"
               transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="Issue">
    <soap:operation
      soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Issue"/>
    <wsdl:input>
      <wsp:PolicyReference
        URI="#Input_policy"/>
      <soap:body use="literal"/>
    </wsdl:input>
    <wsdl:output>
      <wsp:PolicyReference
        URI="#Output_policy"/>
      <soap:body use="literal"/>
    </wsdl:output>
  </wsdl:operation>
  <wsdl:operation name="Validate">
    <soap:operation
      soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Validate"/>
    <wsdl:input>
      <wsp:PolicyReference
        URI="#Input_policy"/>
      <soap:body use="literal"/>
    </wsdl:input>
    <wsdl:output>
      <wsp:PolicyReference
        URI="#Output_policy"/>
      <soap:body use="literal"/>
    </wsdl:output>
  </wsdl:operation>
  <wsdl:operation name="Cancel">
    <soap:operation
      soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Cancel"/>
    <wsdl:input>
      <soap:body use="literal"/>
    </wsdl:input>
    <wsdl:output>
      <soap:body use="literal"/>
    </wsdl:output>
```

```
    </wsdl:operation>
    <wsdl:operation name="Renew">
      <soap:operation
        soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Renew"/>
      <wsdl:input>
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="KeyExchangeToken">
      <soap:operation
        soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/KeyExchangeToken"/>
      <wsdl:input>
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="RequestCollection">
      <soap:operation
        soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/RequestCollection"/>
      <wsdl:input>
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>

  <wsdl:service name="SecurityTokenService">
    <wsdl:port name="UT_Port" binding="tns:UT_Binding">
      <soap:address location="http://localhost:8080/SecurityTokenService/UT"/>
    </wsdl:port>
  </wsdl:service>

  <wsp:Policy wsu:Id="UT_policy">
    <wsp:ExactlyOne>
      <wsp:All>
        <wsap10:UsingAddressing/>
        <sp:SymmetricBinding
          xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
          <wsp:Policy>
            <sp:ProtectionToken>
              <wsp:Policy>
                <sp:X509Token

sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/Never">
                  <wsp:Policy>
                    <sp:RequireDerivedKeys/>
                    <sp:RequireThumbprintReference/>
                    <sp:WssX509V3Token10/>
                  </wsp:Policy>
                </sp:X509Token>
              </wsp:Policy>
```

```
            </sp:ProtectionToken>
            <sp:AlgorithmSuite>
              <wsp:Policy>
                <sp:Basic256/>
              </wsp:Policy>
            </sp:AlgorithmSuite>
            <sp:Layout>
              <wsp:Policy>
                <sp:Lax/>
              </wsp:Policy>
            </sp:Layout>
            <sp:IncludeTimestamp/>
            <sp:EncryptSignature/>
            <sp:OnlySignEntireHeadersAndBody/>
          </wsp:Policy>
        </sp:SymmetricBinding>
        <sp:SignedSupportingTokens
          xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
          <wsp:Policy>
            <sp:UsernameToken

sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/AlwaysToRe
<wsp:Policy>
                <sp:WssUsernameToken10/>
              </wsp:Policy>
            </sp:UsernameToken>
          </wsp:Policy>
        </sp:SignedSupportingTokens>
        <sp:Wss11
          xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
          <wsp:Policy>
            <sp:MustSupportRefKeyIdentifier/>
            <sp:MustSupportRefIssuerSerial/>
            <sp:MustSupportRefThumbprint/>
            <sp:MustSupportRefEncryptedKey/>
          </wsp:Policy>
        </sp:Wss11>
        <sp:Trust13
          xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
          <wsp:Policy>
            <sp:MustSupportIssuedTokens/>
            <sp:RequireClientEntropy/>
            <sp:RequireServerEntropy/>
          </wsp:Policy>
        </sp:Trust13>
      </wsp:All>
    </wsp:ExactlyOne>
  </wsp:Policy>

  <wsp:Policy wsu:Id="Input_policy">
    <wsp:ExactlyOne>
      <wsp:All>
        <sp:SignedParts
          xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
          <sp:Body/>
          <sp:Header Name="To"
                     Namespace="http://www.w3.org/2005/08/addressing"/>
          <sp:Header Name="From"
```

```
                    Namespace="http://www.w3.org/2005/08/addressing"/>
          <sp:Header Name="FaultTo"
                    Namespace="http://www.w3.org/2005/08/addressing"/>
          <sp:Header Name="ReplyTo"
                    Namespace="http://www.w3.org/2005/08/addressing"/>
          <sp:Header Name="MessageID"
                    Namespace="http://www.w3.org/2005/08/addressing"/>
          <sp:Header Name="RelatesTo"
                    Namespace="http://www.w3.org/2005/08/addressing"/>
          <sp:Header Name="Action"
                    Namespace="http://www.w3.org/2005/08/addressing"/>
        </sp:SignedParts>
      </wsp:All>
    </wsp:ExactlyOne>
  </wsp:Policy>

  <wsp:Policy wsu:Id="Output_policy">
    <wsp:ExactlyOne>
      <wsp:All>
        <sp:SignedParts
          xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
          <sp:Body/>
          <sp:Header Name="To"
                    Namespace="http://www.w3.org/2005/08/addressing"/>
          <sp:Header Name="From"
                    Namespace="http://www.w3.org/2005/08/addressing"/>
          <sp:Header Name="FaultTo"
                    Namespace="http://www.w3.org/2005/08/addressing"/>
          <sp:Header Name="ReplyTo"
                    Namespace="http://www.w3.org/2005/08/addressing"/>
          <sp:Header Name="MessageID"
                    Namespace="http://www.w3.org/2005/08/addressing"/>
          <sp:Header Name="RelatesTo"
                    Namespace="http://www.w3.org/2005/08/addressing"/>
          <sp:Header Name="Action"
                    Namespace="http://www.w3.org/2005/08/addressing"/>
        </sp:SignedParts>
      </wsp:All>
    </wsp:ExactlyOne>
  </wsp:Policy>

</wsdl:definitions>
```

STS's implementation class

The Apache CXF's STS, SecurityTokenServiceProvider, is a web service provider that is compliant with the protocols and functionality defined by the WS-Trust specification. It has a modular architecture. Many of its components are configurable or replaceable and there are many optional features that are enabled by implementing and configuring plug-ins. Users can customize their own STS by extending from SecurityTokenServiceProvider and overriding the default settings. Extensive information about the CXF's STS configurable and pluggable components can be found here.

This STS implementation class, SampleSTSHolderOfKey, is a POJO that extends from SecurityTokenServiceProvider. Note that the class is defined with a WebServiceProvider annotation and not a WebService annotation. This annotation defines the service as a Provider-based endpoint, meaning it supports a more messaging-oriented approach to Web services. In particular, it signals that the exchanged messages will be XML documents of some type. SecurityTokenServiceProvider is an implementation of the javax.xml.ws.Provider interface. In comparison the WebService annotation defines a (service endpoint interface) SEI-based endpoint which supports message exchange via SOAP envelopes.

As was done in the HolderOfKeyImpl class, the WSS4J annotations EndpointProperties and EndpointProperty are providing endpoint configuration for the CXF runtime. The first EndpointProperty statements declares the Java properties file that contains the (Merlin) crypto configuration information. WSS4J reads this file and extra required information for message handling. The last EndpointProperty statement declares the STSHolderOfKeyCallbackHandler implementation class. It is used to obtain the user's password for the certificates in the keystore file.

In this implementation we are customizing the operations of token issuance and their static properties.

StaticSTSProperties is used to set select properties for configuring resources in the STS. You may think this is a duplication of the settings made with the WSS4J annotations. The values are the same but the underlaying structures being set are different, thus this information must be declared in both places.

The setIssuer setting is important because it uniquely identifies the issuing STS. The issuer string is embedded in issued tokens and, when validating tokens, the STS checks the issuer string value. Consequently, it is important to use the issuer string in a consistent way, so that the STS can recognize the tokens that it has issued.

The setEndpoints call allows the declaration of a set of allowed token recipients by address. The addresses are specified as reg-ex patterns.

TokenIssueOperation has a modular structure. This allows custom behaviors to be injected into the processing of messages. In this case we are overriding the SecurityTokenServiceProvider's default behavior and performing SAML token processing. CXF provides an implementation of a SAMLTokenProvider which we are using rather than writing our own.

Learn more about the SAMLTokenProvider here.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.stsholderofkey;

import org.apache.cxf.annotations.EndpointProperties;
import org.apache.cxf.annotations.EndpointProperty;
import org.apache.cxf.sts.StaticSTSProperties;
import org.apache.cxf.sts.operation.TokenIssueOperation;
import org.apache.cxf.sts.service.ServiceMBean;
import org.apache.cxf.sts.service.StaticService;
import org.apache.cxf.sts.token.provider.SAMLTokenProvider;
import org.apache.cxf.ws.security.sts.provider.SecurityTokenServiceProvider;

import javax.xml.ws.WebServiceProvider;
import java.util.Arrays;
import java.util.LinkedList;
import java.util.List;
```

```
/**
 * User: rsearls
 * Date: 3/14/14
 */
@WebServiceProvider(serviceName = "SecurityTokenService",
   portName = "UT_Port",
   targetNamespace = "http://docs.oasis-open.org/ws-sx/ws-trust/200512/",
   wsdlLocation = "WEB-INF/wsdl/holderofkey-ws-trust-1.4-service.wsdl")
//be sure to have dependency on org.apache.cxf module when on AS7, otherwise Apache CXF
annotations are ignored
@EndpointProperties(value = {
   @EndpointProperty(key = "ws-security.signature.properties", value =
"stsKeystore.properties"),
   @EndpointProperty(key = "ws-security.callback-handler", value =
"org.jboss.test.ws.jaxws.samples.wsse.policy.trust.stsholderofkey.STSHolderOfKeyCallbackHandler")
class SampleSTSHolderOfKey extends SecurityTokenServiceProvider
{

   public SampleSTSHolderOfKey() throws Exception
   {
      super();

      StaticSTSProperties props = new StaticSTSProperties();
      props.setSignatureCryptoProperties("stsKeystore.properties");
      props.setSignatureUsername("mystskey");
      props.setCallbackHandlerClass(STSHolderOfKeyCallbackHandler.class.getName());
      props.setEncryptionCryptoProperties("stsKeystore.properties");
      props.setEncryptionUsername("myservicekey");
      props.setIssuer("DoubleItSTSIssuer");

      List<ServiceMBean> services = new LinkedList<ServiceMBean>();
      StaticService service = new StaticService();
      service.setEndpoints(Arrays.asList(

"https://localhost:(\\d)*/jaxws-samples-wsse-policy-trust-holderofkey/HolderOfKeyService",

"https://\\[::1\\]:(\\d)*/jaxws-samples-wsse-policy-trust-holderofkey/HolderOfKeyService",

"https://\\[0:0:0:0:0:0:0:1\\]:(\\d)*/jaxws-samples-wsse-policy-trust-holderofkey/HolderOfKeyServi
));

      services.add(service);

      TokenIssueOperation issueOperation = new TokenIssueOperation();
      issueOperation.getTokenProviders().add(new SAMLTokenProvider());
      issueOperation.setServices(services);
      issueOperation.setStsProperties(props);
      this.setIssueOperation(issueOperation);

   }
}
```

HolderOfKeyCallbackHandler

STSHolderOfKeyCallbackHandler is a callback handler for the WSS4J Crypto API. It is used to obtain the password for the private key in the keystore. This class enables CXF to retrieve the password of the user name to use for the message signature.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.stsholderofkey;

import org.jboss.wsf.stack.cxf.extensions.security.PasswordCallbackHandler;

import java.util.HashMap;
import java.util.Map;

/**
 * User: rsearls
 * Date: 3/19/14
 */
public class STSHolderOfKeyCallbackHandler extends PasswordCallbackHandler
{
   public STSHolderOfKeyCallbackHandler()
   {
      super(getInitMap());
   }

   private static Map<String, String> getInitMap()
   {
      Map<String, String> passwords = new HashMap<String, String>();
      passwords.put("mystskey", "stskpass");
      passwords.put("alice", "clarinet");
      return passwords;
   }
}
```

Crypto properties and keystore files

WSS4J's Crypto implementation is loaded and configured via a Java properties file that contains Crypto configuration data. The file contains implementation-specific properties such as a keystore location, password, default alias and the like. This application is using the Merlin implementation. File stsKeystore.properties contains this information.

File servicestore.jks, is a Java KeyStore (JKS) repository. It contains self signed certificates for myservicekey and mystskey. *Self signed certificates are not appropriate for production use.*

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=stsspass
org.apache.ws.security.crypto.merlin.keystore.file=stsstore.jks
```

MANIFEST.MF

When deployed on WildFly, this application requires access to the JBossWs and CXF APIs provided in modules org.jboss.ws.cxf.jbossws-cxf-client and org.apache.cxf. The Apache CXF internals, org.apache.cxf.impl, are needed to build the STS configuration in the SampleSTSHolderOfKey constructor. The dependency statement directs the server to provide them at deployment.

```
Manifest-Version:1.0
Ant-Version: Apache Ant1.8.2
Created-By:1.7.0_25-b15 (Oracle Corporation)
Dependencies: org.jboss.ws.cxf.jbossws-cxf-client,org.apache.cxf.impl
```

Web service requester

This section examines the crucial elements in calling a web service that implements endpoint security as described in the SAML Holder-Of-Key scenario. The components that will be discussed are.

- web service requester's implementation
- ClientCallbackHandler
- Crypto properties and keystore files

Web service requester Implementation

The ws-requester, the client, uses standard procedures for creating a reference to the web service. To address the endpoint security requirements, the web service's "Request Context" is configured with the information needed in message generation. In addition, the STSClient that communicates with the STS is configured with similar values. Note the key strings ending with a ".it" suffix. This suffix flags these settings as belonging to the STSClient. The internal CXF code assigns this information to the STSClient that is auto-generated for this service call.

There is an alternate method of setting up the STSCLient. The user may provide their own instance of the STSClient. The CXF code will use this object and not auto-generate one. When providing the STSClient in this way, the user must provide a org.apache.cxf.Bus for it and the configuration keys must not have the ".it" suffix. This is used in the ActAs and OnBehalfOf examples.

```
String serviceURL = "https://" + getServerHost() +
":8443/jaxws-samples-wsse-policy-trust-holderofkey/HolderOfKeyService";

final QName serviceName = new
QName("http://www.jboss.org/jbossws/ws-extensions/holderofkeywssecuritypolicy",
"HolderOfKeyService");
final URL wsdlURL = new URL(serviceURL + "?wsdl");
Service service = Service.create(wsdlURL, serviceName);
HolderOfKeyIface proxy = (HolderOfKeyIface) service.getPort(HolderOfKeyIface.class);

Map<String, Object> ctx = ((BindingProvider)proxy).getRequestContext();

// set the security related configuration information for the service "request"
ctx.put(SecurityConstants.CALLBACK_HANDLER, new ClientCallbackHandler());
ctx.put(SecurityConstants.SIGNATURE_PROPERTIES,
  Thread.currentThread().getContextClassLoader().getResource(
  "META-INF/clientKeystore.properties"));
ctx.put(SecurityConstants.ENCRYPT_PROPERTIES,
  Thread.currentThread().getContextClassLoader().getResource(
  "META-INF/clientKeystore.properties"));
ctx.put(SecurityConstants.SIGNATURE_USERNAME, "myclientkey");
ctx.put(SecurityConstants.ENCRYPT_USERNAME, "myservicekey");

//-- Configuration settings that will be transfered to the STSClient
// "alice" is the name provided for the WSS Username. Her password will
// be retreived from the ClientCallbackHander by the STSClient.
ctx.put(SecurityConstants.USERNAME + ".it", "alice");
ctx.put(SecurityConstants.CALLBACK_HANDLER + ".it", new ClientCallbackHandler());
ctx.put(SecurityConstants.ENCRYPT_PROPERTIES + ".it",
  Thread.currentThread().getContextClassLoader().getResource(
  "META-INF/clientKeystore.properties"));
ctx.put(SecurityConstants.ENCRYPT_USERNAME + ".it", "mystskey");
ctx.put(SecurityConstants.STS_TOKEN_USERNAME + ".it", "myclientkey");
ctx.put(SecurityConstants.STS_TOKEN_PROPERTIES + ".it",
  Thread.currentThread().getContextClassLoader().getResource(
  "META-INF/clientKeystore.properties"));
ctx.put(SecurityConstants.STS_TOKEN_USE_CERT_FOR_KEYINFO + ".it", "true");

proxy.sayHello();
```

ClientCallbackHandler

ClientCallbackHandler is a callback handler for the WSS4J Crypto API. It is used to obtain the password for the private key in the keystore. This class enables CXF to retrieve the password of the user name to use for the message signature. Note that "alice" and her password have been provided here. This information is not in the (JKS) keystore but provided in the WildFly security domain. It was declared in file jbossws-users.properties.

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.shared;

import java.io.IOException;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import org.apache.ws.security.WSPasswordCallback;

public class ClientCallbackHandler implements CallbackHandler {

    public void handle(Callback[] callbacks) throws IOException,
            UnsupportedCallbackException {
        for (int i = 0; i < callbacks.length; i++) {
            if (callbacks[i] instanceof WSPasswordCallback) {
                WSPasswordCallback pc = (WSPasswordCallback) callbacks[i];
                if ("myclientkey".equals(pc.getIdentifier())) {
                    pc.setPassword("ckpass");
                    break;
                } else if ("alice".equals(pc.getIdentifier())) {
                    pc.setPassword("clarinet");
                    break;
                } else if ("bob".equals(pc.getIdentifier())) {
                    pc.setPassword("trombone");
                    break;
                } else if ("myservicekey".equals(pc.getIdentifier())) {  // rls test  added for
bearer test
                   pc.setPassword("skpass");
                   break;
                }
            }
        }
    }
}
```

Crypto properties and keystore files
WSS4J's Crypto implementation is loaded and configured via a Java properties file that contains Crypto configuration data. The file contains implementation-specific properties such as a keystore location, password, default alias and the like. This application is using the Merlin implementation. File clientKeystore.properties contains this information.

File clientstore.jks, is a Java KeyStore (JKS) repository. It contains self signed certificates for myservicekey and mystskey. *Self signed certificates are not appropriate for production use.*

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=cspass
org.apache.ws.security.crypto.merlin.keystore.alias=myclientkey
org.apache.ws.security.crypto.merlin.keystore.file=META-INF/clientstore.jks
```

Reliable Messaging

JBoss Web Services inherits full WS-Reliable Messaging capabilities from the underlying Apache CXF implementation. At the time of writing, Apache CXF provides support for the WS-Reliable Messaging 1.0 (February 2005) version of the specification.

Enabling WS-Reliable Messaging

WS-Reliable Messaging is implemented internally in Apache CXF through a set of interceptors that deal with the low level requirements of the reliable messaging protocol. In order for enabling WS-Reliable Messaging, users need to either:

- consume a WSDL contract that specifies proper WS-Reliable Messaging policies / assertions
- manually add / configure the reliable messaging interceptors
- specify the reliable messaging policies in an optional CXF Spring XML descriptor
- specify the Apache CXF reliable messaging feature in an optional CXF Spring XML descriptor

The former approach relies on the Apache CXF WS-Policy engine and is the only portable one. The other approaches are Apache CXF proprietary ones, however they allow for fine-grained configuration of protocol aspects that are not covered by the WS-Reliable Messaging Policy. More details are available in the Apache CXF documentation.

Example

In this example we configure WS-Reliable Messaging endpoint and client through the WS-Policy support.

Endpoint

We go with a contract-first approach, so we start by creating a proper WSDL contract, containing the WS-Reliable Messaging and WS-Addressing policies (the latter is a requirement of the former):

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<wsdl:definitions name="SimpleService"
targetNamespace="http://www.jboss.org/jbossws/ws-extensions/wsrm"
  xmlns:tns="http://www.jboss.org/jbossws/ws-extensions/wsrm"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:wsp="http://www.w3.org/2006/07/ws-policy">

  <wsdl:types>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:tns="http://www.jboss.org/jbossws/ws-extensions/wsrm"
  attributeFormDefault="unqualified" elementFormDefault="unqualified"
  targetNamespace="http://www.jboss.org/jbossws/ws-extensions/wsrm">
<xsd:element name="ping" type="tns:ping"/>
<xsd:complexType name="ping">
<xsd:sequence/>
</xsd:complexType>
<xsd:element name="echo" type="tns:echo"/>
<xsd:complexType name="echo">
<xsd:sequence>
<xsd:element minOccurs="0" name="arg0" type="xsd:string"/>
</xsd:sequence>
</xsd:complexType>
<xsd:element name="echoResponse" type="tns:echoResponse"/>
<xsd:complexType name="echoResponse">
<xsd:sequence>
<xsd:element minOccurs="0" name="return" type="xsd:string"/>
</xsd:sequence>
</xsd:complexType>
</xsd:schema>
  </wsdl:types>
  <wsdl:message name="echoResponse">
    <wsdl:part name="parameters" element="tns:echoResponse">
    </wsdl:part>
  </wsdl:message>
  <wsdl:message name="echo">
    <wsdl:part name="parameters" element="tns:echo">
    </wsdl:part>
  </wsdl:message>
  <wsdl:message name="ping">
    <wsdl:part name="parameters" element="tns:ping">
    </wsdl:part>
  </wsdl:message>
  <wsdl:portType name="SimpleService">
    <wsdl:operation name="ping">
      <wsdl:input name="ping" message="tns:ping">
    </wsdl:input>
    </wsdl:operation>
    <wsdl:operation name="echo">
      <wsdl:input name="echo" message="tns:echo">
    </wsdl:input>
      <wsdl:output name="echoResponse" message="tns:echoResponse">
    </wsdl:output>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="SimpleServiceSoapBinding" type="tns:SimpleService">
    <wsp:Policy>
      <!-- WS-Addressing and basic WS-Reliable Messaging policy assertions -->
```

```
      <wswa:UsingAddressing xmlns:wswa="http://www.w3.org/2006/05/addressing/wsdl"/>
      <wsrmp:RMAssertion xmlns:wsrmp="http://schemas.xmlsoap.org/ws/2005/02/rm/policy"/>
      <!-- ------------------------------------------------------------ -->
    </wsp:Policy>
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="ping">
      <soap:operation soapAction="" style="document"/>
      <wsdl:input name="ping">
        <soap:body use="literal"/>
      </wsdl:input>
    </wsdl:operation>
    <wsdl:operation name="echo">
      <soap:operation soapAction="" style="document"/>
      <wsdl:input name="echo">
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output name="echoResponse">
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="SimpleService">
    <wsdl:port name="SimpleServicePort" binding="tns:SimpleServiceSoapBinding">
      <soap:address location="http://localhost:8080/jaxws-samples-wsrm-api"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

Then we use the *wsconsume* tool to generate both standard JAX-WS client and endpoint.

We provide a basic JAX-WS implementation for the endpoint, nothing special in it:

```
package org.jboss.test.ws.jaxws.samples.wsrm.service;

import javax.jws.Oneway;
import javax.jws.WebMethod;
import javax.jws.WebService;

@WebService
(
   name = "SimpleService",
   serviceName = "SimpleService",
   wsdlLocation = "WEB-INF/wsdl/SimpleService.wsdl",
   targetNamespace = "http://www.jboss.org/jbossws/ws-extensions/wsrm"
)
public class SimpleServiceImpl
{
   @Oneway
   @WebMethod
   public void ping()
   {
      System.out.println("ping()");
   }

   @WebMethod
   public String echo(String s)
   {
      System.out.println("echo(" + s + ")");
      return s;
   }
}
```

Finally we package the generated POJO endpoint together with a basic web.xml the usual way and deploy to the application server. The webservices stack automatically detects the policies and enables WS-Reliable Messaging.

Client

The endpoint advertises his RM capabilities (and requirements) through the published WSDL and the client is required to also enable WS-RM for successfully exchanging messages with the server.

So a regular JAX WS client is enough if the user does not need to tune any specific detail of the RM subsystem.

```
QName serviceName = new QName("http://www.jboss.org/jbossws/ws-extensions/wsrm",
"SimpleService");
URL wsdlURL = new URL("http://localhost:8080/jaxws-samples-wsrm-api?wsdl");
Service service = Service.create(wsdlURL, serviceName);
proxy = (SimpleService)service.getPort(SimpleService.class);
proxy.echo("Hello World!");
```

Additional configuration

Fine-grained tuning of WS-Reliable Messaging engine requires setting up proper RM features and attach them for instance to the client proxy. Here is an example:

```
package org.jboss.test.ws.jaxws.samples.wsrm.client;

//...
import javax.xml.ws.Service;
import org.apache.cxf.ws.rm.feature.RMFeature;
import org.apache.cxf.ws.rm.manager.AcksPolicyType;
import org.apache.cxf.ws.rm.manager.DestinationPolicyType;
import org.jboss.test.ws.jaxws.samples.wsrm.generated.SimpleService;

// ...
Service service = Service.create(wsdlURL, serviceName);

RMFeature feature = new RMFeature();
RMAssertion rma = new RMAssertion();
RMAssertion.BaseRetransmissionInterval bri = new RMAssertion.BaseRetransmissionInterval();
bri.setMilliseconds(4000L);
rma.setBaseRetransmissionInterval(bri);
AcknowledgementInterval ai = new AcknowledgementInterval();
ai.setMilliseconds(2000L);
rma.setAcknowledgementInterval(ai);
feature.setRMAssertion(rma);
DestinationPolicyType dp = new DestinationPolicyType();
AcksPolicyType ap = new AcksPolicyType();
ap.setIntraMessageThreshold(0);
dp.setAcksPolicy(ap);
feature.setDestinationPolicy(dp);

SimpleService proxy = (SimpleService)service.getPort(SimpleService.class, feature);
proxy.echo("Hello World");
```

The same can of course be achieved by factoring the feature into a custom pojo extending `org.apache.cxf.ws.rm.feature.RMFeature` and setting the obtained property in a client configuration:

```
package org.jboss.test.ws.jaxws.samples.wsrm.client;

import org.apache.cxf.ws.rm.feature.RMFeature;
import org.apache.cxf.ws.rm.manager.AcksPolicyType;
import org.apache.cxf.ws.rm.manager.DestinationPolicyType;
import org.apache.cxf.ws.rmp.v200502.RMAssertion;
import org.apache.cxf.ws.rmp.v200502.RMAssertion.AcknowledgementInterval;

public class CustomRMFeature extends RMFeature
{
  public CustomRMFeature() {
    super();
    RMAssertion rma = new RMAssertion();
    RMAssertion.BaseRetransmissionInterval bri = new RMAssertion.BaseRetransmissionInterval();
    bri.setMilliseconds(4000L);
    rma.setBaseRetransmissionInterval(bri);
    AcknowledgementInterval ai = new AcknowledgementInterval();
    ai.setMilliseconds(2000L);
    rma.setAcknowledgementInterval(ai);
    super.setRMAssertion(rma);
    DestinationPolicyType dp = new DestinationPolicyType();
    AcksPolicyType ap = new AcksPolicyType();
    ap.setIntraMessageThreshold(0);
    dp.setAcksPolicy(ap);
    super.setDestinationPolicy(dp);
  }
}
```

... this is how the `jaxws-client-config.xml` descriptor would look:

```
<?xml version="1.0" encoding="UTF-8"?>

<jaxws-config xmlns="urn:jboss:jbossws-jaxws-config:4.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:javaee="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="urn:jboss:jbossws-jaxws-config:4.0 schema/jbossws-jaxws-config_4_0.xsd">

<client-config>
<config-name>Custom Client Config</config-name>
<property>
<property-name>cxf.features</property-name>
<property-value>org.jboss.test.ws.jaxws.samples.wsrm.client.CustomRMFeature</property-value>
</property>
</client-config>

</jaxws-config>
```

... and this is how the client would set the configuration:

```
import org.jboss.ws.api.configuration.ClientConfigUtil;
import org.jboss.ws.api.configuration.ClientConfigurer;

//...
Service service = Service.create(wsdlURL, serviceName);
SimpleService proxy = (SimpleService)service.getPort(SimpleService.class);

ClientConfigurer configurer = ClientConfigUtil.resolveClientConfigurer();
configurer.setConfigProperties(proxy, "META-INF/jaxws-client-config.xml", "Custom Client
Config");
proxy.echo("Hello World!");
```

SOAP over JMS

JBoss Web Services allows communication over the *JMS* transport. The functionality comes from Apache CXF support for the SOAP over Java Message Service 1.0 specification, which is aimed at a set of standards for interoperable transport of *SOAP* messages over *JMS*.

On top of Apache CXF functionalities, the JBossWS integration allows users to deploy WS archives containing both *JMS* and *HTTP* endpoints the same way as they do for basic *HTTP* WS endpoints (in *war* archives). The webservices layer of WildFly takes care of looking for *JMS* enpdoints in the deployed archive and starts them delegating to the Apache CXF core similarly as with *HTTP* endpoints.

Configuring SOAP over JMS

As per specification, the *SOAP over JMS* transport configuration is controlled by proper elements and attributes in the `binding` and `service` elements of the WSDL contract. So a *JMS* endpoint is usually developed using a contract-first approach.

The Apache CXF documentation covers all the details of the supported configurations. The minimum configuration implies:

- setting a proper JMS URI in the `soap:address location` [1]
- providing a JNDI connection factory name to be used for connecting to the queues [2]
- setting the transport binding [3]

```
<wsdl:definitions name="HelloWorldService" targetNamespace="http://org.jboss.ws/jaxws/cxf/jms"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://org.jboss.ws/jaxws/cxf/jms"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soapjms="http://www.w3.org/2010/soapjms/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
...

<wsdl:binding name="HelloWorldServiceSoapBinding" type="tns:HelloWorld">
  <soap:binding style="document" transport="http://www.w3.org/2010/soapjms/"/> <!-- 3 -->
  <wsdl:operation name="echo">
    <soap:operation soapAction="" style="document"/>
    <wsdl:input name="echo">
      <soap:body use="literal"/>
    </wsdl:input>
    <wsdl:output name="echoResponse">
      <soap:body use="literal"/>
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>
<wsdl:service name="HelloWorldService">
  <soapjms:jndiConnectionFactoryName>java:/ConnectionFactory</soapjms:jndiConnectionFactoryName>
<!-- 2 -->
  <wsdl:port binding="tns:HelloWorldServiceSoapBinding" name="HelloWorldImplPort">
    <soap:address location="jms:queue:testQueue"/> <!-- 1 -->
  </wsdl:port>
</wsdl:service>
```

Apache CXF takes care of setting up the JMS transport for endpoint implementations whose `@WebService` annotation points to a port declared for JMS transport as explained above.

JBossWS currently supports POJO endpoints only for JMS transport use. The endpoint classes can be deployed as part of *jar* or *war* archives.

The *web.xml* descriptor in *war* archives doesn't need any entry for JMS endpoints.

Examples

JMS endpoint only deployment

In this example we create a simple endpoint relying on *SOAP over JMS* and deploy it as part of a jar archive.

The endpoint is created using wsconsume tool from a WSDL contract such as:

```
<?xml version='1.0' encoding='UTF-8'?>
<wsdl:definitions name="HelloWorldService" targetNamespace="http://org.jboss.ws/jaxws/cxf/jms"
  xmlns:ns1="http://schemas.xmlsoap.org/soap/http"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://org.jboss.ws/jaxws/cxf/jms"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soapjms="http://www.w3.org/2010/soapjms/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <wsdl:types>
<xs:schema elementFormDefault="unqualified" targetNamespace="http://org.jboss.ws/jaxws/cxf/jms"
version="1.0" xmlns:tns="http://org.jboss.ws/jaxws/cxf/jms"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="echo" type="tns:echo"/>
<xs:element name="echoResponse" type="tns:echoResponse"/>
<xs:complexType name="echo">
    <xs:sequence>
      <xs:element minOccurs="0" name="arg0" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
<xs:complexType name="echoResponse">
    <xs:sequence>
      <xs:element minOccurs="0" name="return" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
  </wsdl:types>
  <wsdl:message name="echoResponse">
    <wsdl:part element="tns:echoResponse" name="parameters">
    </wsdl:part>
  </wsdl:message>
  <wsdl:message name="echo">
    <wsdl:part element="tns:echo" name="parameters">
    </wsdl:part>
  </wsdl:message>
  <wsdl:portType name="HelloWorld">
    <wsdl:operation name="echo">
      <wsdl:input message="tns:echo" name="echo">
    </wsdl:input>
      <wsdl:output message="tns:echoResponse" name="echoResponse">
    </wsdl:output>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="HelloWorldServiceSoapBinding" type="tns:HelloWorld">
    <soap:binding style="document" transport="http://www.w3.org/2010/soapjms/"/>
    <wsdl:operation name="echo">
      <soap:operation soapAction="" style="document"/>
      <wsdl:input name="echo">
        <soap:body use="literal"/>
```

```
      </wsdl:input>
      <wsdl:output name="echoResponse">
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="HelloWorldService">

<soapjms:jndiConnectionFactoryName>java:jms/RemoteConnectionFactory</soapjms:jndiConnectionFactory
<soapjms:jndiInitialContextFactory>org.jboss.naming.remote.client.InitialContextFactory</soapjms:j
<soapjms:jndiURL>http-remoting://myhost:8080</soapjms:jndiURL>
    <wsdl:port binding="tns:HelloWorldServiceSoapBinding" name="HelloWorldImplPort">
      <soap:address location="jms:queue:testQueue"/>
    </wsdl:port>
  </wsdl:service>
  <wsdl:service name="HelloWorldServiceLocal">

<soapjms:jndiConnectionFactoryName>java:/ConnectionFactory</soapjms:jndiConnectionFactoryName>
    <wsdl:port binding="tns:HelloWorldServiceSoapBinding" name="HelloWorldImplPort">
      <soap:address location="jms:queue:testQueue"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

The *HelloWorldImplPort* here is meant for using the *testQueue* that has to be created before deploying the endpoint.

At the time of writing, *java:/ConnectionFactory* is the default connection factory JNDI location on WildFly

For allowing remote JNDI lookup of the connection factory, a specific service (`HelloWorldService`) for remote clients is added to the WSDL. The *java:jms/RemoteConnectionFactory* is the JNDI location of the same connection factory mentioned above, except it's exposed for remote lookup. The `soapjms:jndiInitialContextFactory` and `soap:jmsjndiURL` complete the remote connection configuration, specifying the initial context factory class to use and the JNDI registry address.

Have a look at the application server domain for finding out the configured connection factory JNDI locations.

The endpoint implementation is a basic JAX-WS POJO using @WebService annotation to refer to the consumed contract:

```
package org.jboss.test.ws.jaxws.cxf.jms;

import javax.jws.WebService;

@WebService
(
   portName = "HelloWorldImplPort",
   serviceName = "HelloWorldServiceLocal",
   wsdlLocation = "META-INF/wsdl/HelloWorldService.wsdl",
   endpointInterface = "org.jboss.test.ws.jaxws.cxf.jms.HelloWorld",
   targetNamespace = "http://org.jboss.ws/jaxws/cxf/jms"
)
public class HelloWorldImpl implements HelloWorld
{
   public String echo(String input)
   {
      return input;
   }
}
```

The endpoint implementation references the `HelloWorldServiceLocal` wsdl service, so that the local JNDI connection factory location is used for starting the endpoint on server side.

That's pretty much all. We just need to package the generated service endpoint interface, the endpoint implementation and the WSDL file in a *jar* archive and deploy it:

```
alessio@inuyasha /dati/jbossws/stack/cxf/trunk $ jar -tvf
./modules/testsuite/cxf-tests/target/test-libs/jaxws-cxf-jms-only-deployment.jar
     0 Thu Jun 23 15:18:44 CEST 2011 META-INF/
   129 Thu Jun 23 15:18:42 CEST 2011 META-INF/MANIFEST.MF
     0 Thu Jun 23 15:18:42 CEST 2011 org/
     0 Thu Jun 23 15:18:42 CEST 2011 org/jboss/
     0 Thu Jun 23 15:18:42 CEST 2011 org/jboss/test/
     0 Thu Jun 23 15:18:42 CEST 2011 org/jboss/test/ws/
     0 Thu Jun 23 15:18:42 CEST 2011 org/jboss/test/ws/jaxws/
     0 Thu Jun 23 15:18:42 CEST 2011 org/jboss/test/ws/jaxws/cxf/
     0 Thu Jun 23 15:18:42 CEST 2011 org/jboss/test/ws/jaxws/cxf/jms/
   313 Thu Jun 23 15:18:42 CEST 2011 org/jboss/test/ws/jaxws/cxf/jms/HelloWorld.class
  1173 Thu Jun 23 15:18:42 CEST 2011 org/jboss/test/ws/jaxws/cxf/jms/HelloWorldImpl.class
     0 Thu Jun 23 15:18:40 CEST 2011 META-INF/wsdl/
  3074 Thu Jun 23 15:18:40 CEST 2011 META-INF/wsdl/HelloWorldService.wsdl
```

A dependency on `org.hornetq` module needs to be added in MANIFEST.MF when deploying to WildFly.

```
Manifest-Version: 1.0

Ant-Version: Apache Ant 1.7.1

Created-By: 17.0-b16 (Sun Microsystems Inc.)

Dependencies: org.hornetq
```

A JAX-WS client can interact with the JMS endpoint the usual way:

```
URL wsdlUrl = ...
//start another bus to avoid affecting the one that could already be assigned to the current
thread - optional but highly suggested
Bus bus = BusFactory.newInstance().createBus();
BusFactory.setThreadDefaultBus(bus);
try
{
   QName serviceName = new QName("http://org.jboss.ws/jaxws/cxf/jms", "HelloWorldService");
   Service service = Service.create(wsdlUrl, serviceName);
   HelloWorld proxy = (HelloWorld) service.getPort(new
QName("http://org.jboss.ws/jaxws/cxf/jms", "HelloWorldImplPort"), HelloWorld.class);
   setupProxy(proxy);
   proxy.echo("Hi");
}
finally
{
   bus.shutdown(true);
}
```

The WSDL location URL needs to be retrieved in a custom way, depending on the client application. Given the endpoint is JMS only, there's no automatically published WSDL contract.

in order for performing the remote invocation (which internally goes through remote JNDI lookup of the connection factory), the calling user credentials need to be set into the Apache CXF JMSConduit:

```
private void setupProxy(HelloWorld proxy) {
   JMSConduit conduit = (JMSConduit)ClientProxy.getClient(proxy).getConduit();
   JNDIConfiguration jndiConfig = conduit.getJmsConfig().getJndiConfig();
   jndiConfig.setConnectionUserName("user");
   jndiConfig.setConnectionPassword("password");
   Properties props = conduit.getJmsConfig().getJndiTemplate().getEnvironment();
   props.put(Context.SECURITY_PRINCIPAL, "user");
   props.put(Context.SECURITY_CREDENTIALS, "password");
}
```

Have a look at the WildFly domain and messaging configuration for finding out the actual security requirements. At the time of writing, a user with `guest` role is required and that's internally checked using the `other` security domain.

Of course once the endpoint is exposed over JMS transport, any plain JMS client can also be used to send messages to the webservice endpoint. You can have a look at the SOAP over JMS spec details and code the client similarly to

```
Properties env = new Properties();
env.put(Context.INITIAL_CONTEXT_FACTORY,
"org.jboss.naming.remote.client.InitialContextFactory");
env.put(Context.PROVIDER_URL, "http-remoting://myhost:8080");
env.put(Context.SECURITY_PRINCIPAL, "user");
env.put(Context.SECURITY_CREDENTIALS, "password");
InitialContext context = new InitialContext(env);
QueueConnectionFactory connectionFactory =
(QueueConnectionFactory)context.lookup("jms/RemoteConnectionFactory");
Queue reqQueue = (Queue)context.lookup("jms/queue/test");
Queue resQueue = (Queue)context.lookup("jms/queue/test");
QueueConnection con = connectionFactory.createQueueConnection("user", "password");
QueueSession session = con.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
QueueReceiver receiver = session.createReceiver(resQueue);
ResponseListener responseListener = new ResponseListener(); //a custom response listener...
receiver.setMessageListener(responseListener);
con.start();
TextMessage message = session.createTextMessage(reqMessage);
message.setJMSReplyTo(resQueue);

//setup SOAP-over-JMS properties...
message.setStringProperty("SOAPJMS_contentType", "text/xml");
message.setStringProperty("SOAPJMS_requestURI", "jms:queue:testQueue");

QueueSender sender = session.createSender(reqQueue);
sender.send(message);
sender.close();

...
```

JMS and HTTP endpoints deployment

In this example we create a deployment containing an endpoint that serves over both HTTP and JMS transports.

We from a WSDL contract such as below (please note we've two `binding` / `portType` for the same `service`):

```
<?xml version='1.0' encoding='UTF-8'?>
<wsdl:definitions name="HelloWorldService" targetNamespace="http://org.jboss.ws/jaxws/cxf/jms"
  xmlns:ns1="http://schemas.xmlsoap.org/soap/http"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://org.jboss.ws/jaxws/cxf/jms"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soapjms="http://www.w3.org/2010/soapjms/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <wsdl:types>
<xs:schema elementFormDefault="unqualified" targetNamespace="http://org.jboss.ws/jaxws/cxf/jms"
version="1.0"
  xmlns:tns="http://org.jboss.ws/jaxws/cxf/jms" xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="echo" type="tns:echo"/>
<xs:element name="echoResponse" type="tns:echoResponse"/>
<xs:complexType name="echo">
    <xs:sequence>
      <xs:element minOccurs="0" name="arg0" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
<xs:complexType name="echoResponse">
    <xs:sequence>
      <xs:element minOccurs="0" name="return" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
  </wsdl:types>
  <wsdl:message name="echoResponse">
    <wsdl:part element="tns:echoResponse" name="parameters">
    </wsdl:part>
  </wsdl:message>
  <wsdl:message name="echo">
    <wsdl:part element="tns:echo" name="parameters">
    </wsdl:part>
  </wsdl:message>
  <wsdl:portType name="HelloWorld">
    <wsdl:operation name="echo">
      <wsdl:input message="tns:echo" name="echo">
    </wsdl:input>
      <wsdl:output message="tns:echoResponse" name="echoResponse">
    </wsdl:output>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="HelloWorldServiceSoapBinding" type="tns:HelloWorld">
    <soap:binding style="document" transport="http://www.w3.org/2010/soapjms/"/>
    <wsdl:operation name="echo">
      <soap:operation soapAction="" style="document"/>
      <wsdl:input name="echo">
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output name="echoResponse">
```

```
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:binding name="HttpHelloWorldServiceSoapBinding" type="tns:HelloWorld">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="echo">
      <soap:operation soapAction="" style="document"/>
      <wsdl:input name="echo">
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output name="echoResponse">
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="HelloWorldService">

<soapjms:jndiConnectionFactoryName>java:jms/RemoteConnectionFactory</soapjms:jndiConnectionFactory
<soapjms:jndiInitialContextFactory>org.jboss.naming.remote.client.InitialContextFactory</soapjms:j
<soapjms:jndiURL>http-remoting://localhost:8080</soapjms:jndiURL>
    <wsdl:port binding="tns:HelloWorldServiceSoapBinding" name="HelloWorldImplPort">
      <soap:address location="jms:queue:testQueue"/>
    </wsdl:port>
    <wsdl:port binding="tns:HttpHelloWorldServiceSoapBinding" name="HttpHelloWorldImplPort">
      <soap:address location="http://localhost:8080/jaxws-cxf-jms-http-deployment"/>
    </wsdl:port>
  </wsdl:service>
  <wsdl:service name="HelloWorldServiceLocal">

<soapjms:jndiConnectionFactoryName>java:/ConnectionFactory</soapjms:jndiConnectionFactoryName>
    <wsdl:port binding="tns:HelloWorldServiceSoapBinding" name="HelloWorldImplPort">
      <soap:address location="jms:queue:testQueue"/>
    </wsdl:port>
</wsdl:definitions>
```

The same considerations of the previous example regarding the JMS queue and JNDI connection factory still apply.

Here we can implement the endpoint in multiple ways, either with a common implementation class that's extended by the JMS and HTTP ones, or keep the two implementation classes independent and just have them implement the same service endpoint interface:

```
package org.jboss.test.ws.jaxws.cxf.jms_http;

import javax.jws.WebService;

@WebService
(
   portName = "HelloWorldImplPort",
   serviceName = "HelloWorldServiceLocal",
   wsdlLocation = "WEB-INF/wsdl/HelloWorldService.wsdl",
   endpointInterface = "org.jboss.test.ws.jaxws.cxf.jms_http.HelloWorld",
   targetNamespace = "http://org.jboss.ws/jaxws/cxf/jms"
)
public class HelloWorldImpl implements HelloWorld
{
   public String echo(String input)
   {
      System.out.println("input: " + input);
      return input;
   }
}
```

```
package org.jboss.test.ws.jaxws.cxf.jms_http;

import javax.jws.WebService;

@WebService
(
   portName = "HttpHelloWorldImplPort",
   serviceName = "HelloWorldService",
   wsdlLocation = "WEB-INF/wsdl/HelloWorldService.wsdl",
   endpointInterface = "org.jboss.test.ws.jaxws.cxf.jms_http.HelloWorld",
   targetNamespace = "http://org.jboss.ws/jaxws/cxf/jms"
)
public class HttpHelloWorldImpl implements HelloWorld
{
   public String echo(String input)
   {
      System.out.println("input (http): " + input);
      return "(http) " + input;
   }
}
```

Both classes are packaged together the service endpoint interface and the WSDL file in a *war* archive:

```
alessio@inuyasha /dati/jbossws/stack/cxf/trunk $ jar -tvf
./modules/testsuite/cxf-spring-tests/target/test-libs/jaxws-cxf-jms-http-deployment.war
     0 Thu Jun 23 15:18:44 CEST 2011 META-INF/
   129 Thu Jun 23 15:18:42 CEST 2011 META-INF/MANIFEST.MF
     0 Thu Jun 23 15:18:44 CEST 2011 WEB-INF/
   569 Thu Jun 23 15:18:40 CEST 2011 WEB-INF/web.xml
     0 Thu Jun 23 15:18:44 CEST 2011 WEB-INF/classes/
     0 Thu Jun 23 15:18:42 CEST 2011 WEB-INF/classes/org/
     0 Thu Jun 23 15:18:42 CEST 2011 WEB-INF/classes/org/jboss/
     0 Thu Jun 23 15:18:42 CEST 2011 WEB-INF/classes/org/jboss/test/
     0 Thu Jun 23 15:18:42 CEST 2011 WEB-INF/classes/org/jboss/test/ws/
     0 Thu Jun 23 15:18:42 CEST 2011 WEB-INF/classes/org/jboss/test/ws/jaxws/
     0 Thu Jun 23 15:18:42 CEST 2011 WEB-INF/classes/org/jboss/test/ws/jaxws/cxf/
     0 Thu Jun 23 15:18:42 CEST 2011 WEB-INF/classes/org/jboss/test/ws/jaxws/cxf/jms_http/
   318 Thu Jun 23 15:18:42 CEST 2011
WEB-INF/classes/org/jboss/test/ws/jaxws/cxf/jms_http/HelloWorld.class
  1192 Thu Jun 23 15:18:42 CEST 2011
WEB-INF/classes/org/jboss/test/ws/jaxws/cxf/jms_http/HelloWorldImpl.class
  1246 Thu Jun 23 15:18:42 CEST 2011
WEB-INF/classes/org/jboss/test/ws/jaxws/cxf/jms_http/HttpHelloWorldImpl.class
     0 Thu Jun 23 15:18:40 CEST 2011 WEB-INF/wsdl/
  3068 Thu Jun 23 15:18:40 CEST 2011 WEB-INF/wsdl/HelloWorldService.wsdl
```

A trivial web.xml descriptor is also included to trigger the HTTP endpoint publish:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
  version="2.4">
  <servlet>
    <servlet-name>EndpointServlet</servlet-name>
    <servlet-class>org.jboss.test.ws.jaxws.cxf.jms_http.HttpHelloWorldImpl</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>EndpointServlet</servlet-name>
    <url-pattern>/*</url-pattern>
  </servlet-mapping>
</web-app>
```

Here too the MANIFEST.MF needs to declare a dependency on *org.hornetq* module when deploying to WildFly.

Finally, the JAX-WS client can ineract with both JMS and HTTP endpoints as usual:

```
//start another bus to avoid affecting the one that could already be assigned to current thread
- optional but highly suggested
Bus bus = BusFactory.newInstance().createBus();
BusFactory.setThreadDefaultBus(bus);
try
{
   QName serviceName = new QName("http://org.jboss.ws/jaxws/cxf/jms", "HelloWorldService");
   Service service = Service.create(wsdlUrl, serviceName);

   //JMS test
   HelloWorld proxy = (HelloWorld) service.getPort(new
QName("http://org.jboss.ws/jaxws/cxf/jms", "HelloWorldImplPort"), HelloWorld.class);
   setupProxy(proxy);
   proxy.echo("Hi");
   //HTTP test
   HelloWorld httpProxy = (HelloWorld) service.getPort(new
QName("http://org.jboss.ws/jaxws/cxf/jms", "HttpHelloWorldImplPort"), HelloWorld.class);
   httpProxy.echo("Hi");
}
finally
{
   bus.shutdown(true);
}
```

Use of Endpoint.publish() API

An alternative to deploying an archive containing JMS endpoints is in starting them directly using the JAX-WS `Endpoint.publish(..)` API.

That's as easy as doing:

```
Object implementor = new HelloWorldImpl();
Endpoint ep = Endpoint.publish("jms:queue:testQueue", implementor);
try
{
   //use or let others use the endpoint
}
finally
{
   ep.stop();
}
```

where `HelloWorldImpl` is a POJO endpoint implementation referencing a JMS *port* in a given WSDL contract, as explained in the previous examples.

The main difference among the deployment approach is in the direct control and responsibility over the endpoint lifecycle (*start/publish* and *stop*).

HTTP Proxy

The HTTP Proxy related functionalities of JBoss Web Services are provided by the Apache CXF http transport layer.

The suggested configuration mechanism when running JBoss Web Services is explained below; for further information please refer to the Apache CXF documentation.

Configuration

The HTTP proxy configuration for a given JAX-WS client can be set in the following ways:

- through the `http.proxyHost` and `http.proxyPort` system properties, or
- leveraging the `org.apache.cxf.transport.http.HTTPConduit` options

The former is a JVM level configuration; for instance, assuming the http proxy is currently running at http://localhost:9934, here is the setup:

```
System.getProperties().setProperty("http.proxyHost", "localhost");
System.getProperties().setProperty("http.proxyPort", 9934);
```

The latter is a client stub/port level configuration: the setup is performed on the `HTTPConduit` object that's part of the Apache CXF `Client` abstraction.

```
import org.apache.cxf.configuration.security.ProxyAuthorizationPolicy;
import org.apache.cxf.endpoint.Client;
import org.apache.cxf.frontend.ClientProxy;
import org.apache.cxf.transport.http.HTTPConduit;
import org.apache.cxf.transports.http.configuration.HTTPClientPolicy;
import org.apache.cxf.transports.http.configuration.ProxyServerType;
...

Service service = Service.create(wsdlURL, new QName("http://org.jboss.ws/jaxws/cxf/httpproxy",
"HelloWorldService"));
HelloWorld port = (HelloWorld) service.getPort(new
QName("http://org.jboss.ws/jaxws/cxf/httpproxy", "HelloWorldImplPort"), HelloWorld.class);

Client client = ClientProxy.getClient(port);
HTTPConduit conduit = (HTTPConduit)client.getConduit();
ProxyAuthorizationPolicy policy = new ProxyAuthorizationPolicy();
policy.setAuthorizationType("Basic");
policy.setUserName(PROXY_USER);
policy.setPassword(PROXY_PWD);
conduit.setProxyAuthorization(policy);

port.echo("Foo");
```

The `ProxyAuthorizationPolicy` also allows for setting the authotization type as well as the username / password to be used.

Speaking of authorization and authentication, please note that the JDK already features the `java.net.Authenticator` facility, which is used whenever opening a connection to a given URL requiring a http proxy. Users might want to set a custom Authenticator for instance when needing to read WSDL contracts before actually calling into the JBoss Web Services / Apache CXF code; here is an example:

```
import java.net.Authenticator;
import java.net.PasswordAuthentication;
...
public class ProxyAuthenticator extends Authenticator
{
   private String user, password;

   public ProxyAuthenticator(String user, String password)
   {
      this.user = user;
      this.password = password;
   }

   protected PasswordAuthentication getPasswordAuthentication()
   {
      return new PasswordAuthentication(user, password.toCharArray());
   }
}

...

Authenticator.setDefault(new ProxyAuthenticator(PROXY_USER, PROXY_PWD));
```

Discovery

Apache CXF includes support for *Web Services Dynamic Discovery* (WS-Discovery), which is a protocol to enable dynamic discovery of services available on the local network. The protocol implies using a `UDP` based multicast transport to announce new services and probe for existing services. A managed mode where a discovery proxy is used to reduce the amount of required multicast traffic is also covered by the protocol.

JBossWS integrates the *WS-Discovery* functionalities provided by Apache CXF into the application server.

Enabling WS-Discovery

Apache CXF enables *WS-Discovery* depending on the availability of its runtime component; given that's always shipped in the application server, JBossWS integration requires using the `cxf.ws-discovery.enabled` property usage for enabling *WS-Discovery* for a given deployment. By default *WS-Discovery* is disabled on the application server. Below is an example of *jboss-webservices.xml* descriptor to be used for enabling *WS-Discovery*:

```
<webservices xmlns="http://www.jboss.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  version="1.2" xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee">

  <property>
    <name>cxf.ws-discovery.enabled</name>
    <value>true</value>
  </property>

</webservices>
```

By default, a *WS-Discovery* service endpoint (SOAP-over-UDP bound) will be started the first time a WS-Discovery enabled deployment is processed on the application server. Every ws endpoint belonging to *WS-Discovery* enabled deployments will be automatically registered into such a *WS-Discovery* service endpoint (`Hello` messages). The service will reply to `Probe` and `Resolve` messages received on `UDP` port `3702` (including multicast messages sent to `IPv4` address `239.255.255.250`, as per specification). Endpoints will eventually be automatically unregistered using `Bye` messages upon undeployment.

Probing services

Apache CXF comes with a *WS-Discovery* API that can be used to probe / resolve services. When running in-container, a JBoss module dependency to the `org.apache.cxf.impl` module is to be set to have access to *WS-Discovery* client functionalities.

The org.apache.cxf.ws.discovery.WSDiscoveryClient class provides the *probe* and *resolve* methods which also accepts filters on scopes. Users can rely on them for locating available endpoints on the network. Please have a look at the JBossWS testsuite which includes a sample on CXF WS-Discovery usage.

Policy

Apache CXF WS-Policy support

JBossWS policy support rely on the Apache CXF WS-Policy framework, which is compliant with the Web Services Policy 1.5 - Framework and Web Services Policy 1.5 - Attachment specifications.
Users can work with policies in different ways:

- by adding policy assertions to wsdl contracts and letting the runtime consume them and behave accordingly;
- by specifying endpoint policy attachments using either CXF annotations or features.

Of course users can also make direct use of the Apache CXF policy framework, defining custom assertions, etc.

Finally, JBossWS provides some additional annotations for simplified policy attachment.

Contract-first approach

WS-Policies can be attached and referenced in wsdl elements (the specifications describe all possible alternatives). Apache CXF automatically recognizes, reads and uses policies defined in the wsdl.

Users should hence develop endpoints using the *contract-first* approach, that is explicitly providing the contract for their services. Here is a excerpt taken from a wsdl including a WS-Addressing policy:

```
<wsdl:definitions name="Foo" targetNamespace="http://ws.jboss.org/foo"
...
<wsdl:service name="FooService">
    <wsdl:port binding="tns:FooBinding" name="FooPort">
        <soap:address location="http://localhost:80800/foo"/>
        <wsp:Policy xmlns:wsp="http://www.w3.org/ns/ws-policy">
             <wsam:Addressing xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata">
                 <wsp:Policy/>
              </wsam:Addressing>
         </wsp:Policy>
    </wsdl:port>
</wsdl:service>
</wsdl:definitions>
```

Of course, CXF also acts upon policies specified in wsdl documents consumed on client side.

Code-first approach

For those preferring code-first (java-first) endpoint development, Apache CXF comes with `org.apache.cxf.annotations.Policy` and `org.apache.cxf.annotations.Policies` annotations to be used for attaching policy fragments to the wsdl generated at deploy time.

Here is an example of a code-first endpoint including @Policy annotation:

```
import javax.jws.WebService;
import org.apache.cxf.annotations.Policy;

@WebService(portName = "MyServicePort",
            serviceName = "MyService",
            name = "MyServiceIface",
            targetNamespace = "http://www.jboss.org/jbossws/foo")
@Policy(placement = Policy.Placement.BINDING, uri = "JavaFirstPolicy.xml")
public class MyServiceImpl {
   public String sayHello() {
      return "Hello World!";
   }
}
```

The referenced descriptor is to be added to the deployment and will include the policy to be attached; the attachment position in the contracts is defined through the `placement` attribute. Here is a descriptor example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<wsp:Policy wsu:Id="MyPolicy" xmlns:wsp="http://www.w3.org/ns/ws-policy"

xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">
    <wsp:ExactlyOne>
        <wsp:All>
            <sp:SupportingTokens
xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
                <wsp:Policy>
                    <sp:UsernameToken
sp:IncludeToken="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy/IncludeToken/AlwaysToRecipie
<wsp:Policy>
                            <sp:WssUsernameToken10/>
                        </wsp:Policy>
                    </sp:UsernameToken>
                </wsp:Policy>
            </sp:SupportingTokens>
        </wsp:All>
    </wsp:ExactlyOne>
</wsp:Policy>
```

JBossWS additions

Policy sets

Both approaches above require users to actually write their policies' assertions; while this offer great flexibility and control of the actual contract, providing the assertions might end up being quite a challenging task for complex policies. For this reason, the JBossWS integration provides *policy sets*, which are basically pre-defined groups of policy assertions corresponding to well known / common needs. Each set has a label allowing users to specify it in the `@org.jboss.ws.api.annotation.PolicySets` annotation to have the policy assertions for that set attached to the annotated endpoint. Multiple labels can also be specified. Here is an example of the @PolicySets annotation on a service endpoint interface:

```
import javax.jws.WebService;
import org.jboss.ws.api.annotation.PolicySets;

@WebService(name = "EndpointTwo", targetNamespace = "http://org.jboss.ws.jaxws.cxf/jbws3648")
@PolicySets({"WS-RM_Policy_spec_example", "WS-SP-EX223_WSS11_Anonymous_X509_Sign_Encrypt",
"WS-Addressing"})
public interface EndpointTwo
{
   String echo(String input);
}
```

The three sets specified in @PolicySets will cause the wsdl generated for the endpoint having this interface to be enriched with some policy assertions for WS-RM, WS-Security and WS-Addressing.

The labels' list of known sets is stored in the `META-INF/policies/org.jboss.wsf.stack.cxf.extensions.policy.PolicyAttachmentStor` file within the `jbossws-cxf-client.jar` (`org.jboss.ws.cxf:jbossws-cxf-client` maven artifact). Actual policy fragments for each set are also stored in the same artifact at `META-INF/policies/<set-label>-<attachment-position>.xml`.

Here is a list of the available policy sets:

Label	Description
WS-Addressing	Basic WS-Addressing policy
WS-RM_Policy_spec_example	The basic WS-RM policy example in the WS-RM specification
WS-SP-EX2121_SSL_UT_Supporting_Token	The group of policy assertions used in the section 2.1.2.1 example of the WS-Security Policy Examples 1.0 specification

WS-SP-EX213_WSS10_UT_Mutual_Auth_X509_Sign_Encrypt	The group of policy assertions used in the section 2.1.3 example of the WS-Security Policy Examples 1.0 specification
WS-SP-EX214_WSS11_User_Name_Cert_Sign_Encrypt	The group of policy assertions used in the section 2.1.4 example of the WS-Security Policy Examples 1.0 specification
WS-SP-EX221_WSS10_Mutual_Auth_X509_Sign_Encrypt	The group of policy assertions used in the section 2.2.1 example of the WS-Security Policy Examples 1.0 specification
WS-SP-EX222_WSS10_Mutual_Auth_X509_Sign_Encrypt	The group of policy assertions used in the section 2.2.2 example of the WS-Security Policy Examples 1.0 specification
WS-SP-EX223_WSS11_Anonymous_X509_Sign_Encrypt	The group of policy assertions used in the section 2.2.3 example of the WS-Security Policy Examples 1.0 specification
WS-SP-EX224_WSS11_Mutual_Auth_X509_Sign_Encrypt	The group of policy assertions used in the section 2.2.4 example of the WS-Security Policy Examples 1.0 specification

AsymmetricBinding_X509v1_TripleDesRsa15_EncryptBeforeSigning_ProtectTokens	A WS-Security policy for asymmetric binding (encrypt before signing) using X.509v1 tokens, 3DES + RSA 1.5 algorithms and with token protections enabled
AsymmetricBinding_X509v1_GCM256OAEP_ProtectTokens	The same as before, but using custom Apache CXF algorithm suite including GCM 256 + RSA OAEP algorithms

Always verify the contents of the generated wsdl contract, as policy sets are potentially subject to updates between JBossWS releases. This is especially important when dealing with security related policies; the provided sets are to be considered as convenient configuration options only; users remain responsible for the policies in their contracts.

The `org.jboss.wsf.stack.cxf.extensions.policy.Constants` interface has convenient String constants for the available policy set labels.

If you feel a new set should be added, just propose it by writing the user forum!

Published WSDL customization

- Endpoint address rewrite
- System property references

Endpoint address rewrite

JBossWS supports the rewrite of the `<soap:address>` element of endpoints published in WSDL contracts. This feature is useful for controlling the server address that is advertised to clients for each endpoint. The rewrite mechanism is configured at server level through a set of elements in the webservices subsystem of the WildFly management model. Please refer to the container documentation for details on the options supported in the selected container version. Below is a list of the elements available in the latest WildFly sources:

Name	Type	Description
modify-wsdl-address	boolean	This boolean enables and disables the address rewrite functionality. When modify-wsdl-address is set to true and the content of <soap:address> is a valid URL, JBossWS will rewrite the URL using the values of wsdl-host and wsdl-port or wsdl-secure-port. When modify-wsdl-address is set to false and the content of <soap:address> is a valid URL, JBossWS will not rewrite the URL. The <soap:address> URL will be used. When the content of <soap:address> is not a valid URL, JBossWS will rewrite it no matter what the setting of modify-wsdl-address. If modify-wsdl-address is set to true and wsdl-host is not defined or explicitly set to '`jbossws.undefined.host`' the content of <soap:address> URL is use. JBossWS uses the requester's host when rewriting the <soap:address> When modify-wsdl-address is not defined JBossWS uses a default value of true.
wsdl-host	string	The hostname / IP address to be used for rewriting `<soap:address>`. If `wsdl-host` is set to `jbossws.undefined.host`, JBossWS uses the requester's host when rewriting the `<soap:address>` When wsdl-host is not defined JBossWS uses a default value of '`jbossws.undefined.host`'.
wsdl-port	int	Set this property to explicitly define the HTTP port that will be used for rewriting the SOAP address. Otherwise the HTTP port will be identified by querying the list of installed HTTP connectors.
wsdl-secure-port	int	Set this property to explicitly define the HTTPS port that will be used for rewriting the SOAP address. Otherwise the HTTPS port will be identified by querying the list of installed HTTPS connectors.
wsdl-uri-scheme	string	This property explicitly sets the URI scheme to use for rewriting `<soap:address>` . Valid values are `http` and `https`. This configuration overrides scheme computed by processing the endpoint (even if a transport guarantee is specified). The provided values for `wsdl-port` and `wsdl-secure-port` (or their default values) are used depending on specified scheme.
wsdl-path-rewrite-rule	string	This string defines a SED substitution command (e.g., 's/regexp/replacement/g') that JBossWS executes against the path component of each <soap:address> URL published from the server. When wsdl-path-rewrite-rule is not defined, JBossWS retains the original path component of each <soap:address> URL. When 'modify-wsdl-address' is set to "false" this element is ignored.

Additionally, users can override the server level configuration by requesting a specific rewrite behavior for a given endpoint deployment. That is achieved by setting one of the following properties within a *jboss-webservices.xml* descriptor:

Property	Corresponding server option
wsdl.soapAddress.rewrite.modify-wsdl-address	modify-wsdl-address
wsdl.soapAddress.rewrite.wsdl-host	wsdl-host
wsdl.soapAddress.rewrite.wsdl-port	wsdl-port
wsdl.soapAddress.rewrite.wsdl-secure-port	wsdl-secure-port
wsdl.soapAddress.rewrite.wsdl-path-rewrite-rule	wsdl-path-rewrite-rule
wsdl.soapAddress.rewrite.wsdl-uri-scheme	wsdl-uri-scheme

Here is an example of partial overriding of the default configuration for a specific deployment:

```
<?xml version="1.1" encoding="UTF-8"?>
<webservices version="1.2"
  xmlns="http://www.jboss.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee">
  <property>
    <name>wsdl.soapAddress.rewrite.wsdl-uri-scheme</name>
    <value>https</value>
  </property>
  <property>
    <name>wsdl.soapAddress.rewrite.wsdl-host</name>
    <value>foo</value>
  </property>
</webservices>
```

System property references

System property references wrapped within "@" characters are expanded when found in WSDL attribute and element values. This allows for instance including multiple WS-Policy declarations in the contract and selecting the policy to use depending on a server wide system property; here is an example:

```
<wsdl:definitions ...>
  ...
  <wsdl:binding name="ServiceOneSoapBinding" type="tns:EndpointOne">
    ...
    <wsp:PolicyReference URI="#@org.jboss.wsf.test.JBWS3628TestCase.policy@"/>
    <wsdl:operation name="echo">
      ...
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="ServiceOne">
    <wsdl:port binding="tns:ServiceOneSoapBinding" name="EndpointOnePort">
      <soap:address location="http://localhost:8080/jaxws-cxf-jbws3628/ServiceOne"/>
    </wsdl:port>
  </wsdl:service>

  <wsp:Policy
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
xmlns:wsp="http://www.w3.org/ns/ws-policy" wsu:Id="WS-RM_Policy">
 <wsrmp:RMAssertion xmlns:wsrmp="http://schemas.xmlsoap.org/ws/2005/02/rm/policy">
          ...
   </wsrmp:RMAssertion>
  </wsp:Policy>

  <wsp:Policy
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
xmlns:wsp="http://www.w3.org/ns/ws-policy"
      xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata" wsu:Id="WS-Addressing_policy">
    <wsam:Addressing>
      <wsp:Policy/>
    </wsam:Addressing>
  </wsp:Policy>
</wsdl:definitions>
```

If the ***org.jboss.wsf.test.JBWS3628TestCase.policy*** system property is defined and set to "***WS-Addressing_policy***", WS-Addressing will be enabled for the endpoint defined by the contract above.

6.35.4 JBoss Modules and WS applications

- Setting module dependencies
  - Using MANIFEST.MF
    - Using JAXB
    - Using Apache CXF
    - Client side WS aggregation module
    - Annotation scanning
  - Using jboss-deployment-descriptor.xml

The JBoss Web Services functionalities are provided by a given set of modules / libraries installed on WildFly, which are organized into JBoss Modules modules. In particular the *org.jboss.as.webservices.** and *org.jboss.ws.** modules belong to the JBossWS - WildFly integration. Users should not need to change anything in them.

While users are of course allowed to provide their own modules for their custom needs, below is a brief collection of suggestions and hints around modules and webservices development on WildFly.

Setting module dependencies

On WildFly the user deployment classloader does not have any visibility over JBoss internals; so for instance you can't *directly* use JBossWS *implementation* classes unless you explicitly set a dependency to the corresponding module. As a consequence, users need to declare the module dependencies they want to be added to their deployment.

The JBoss Web Services APIs are always available by default whenever the webservices subsystem is available on AS7. So users just use them, no need for explicit dependencies declaration for those modules.

Using MANIFEST.MF

The convenient method for configuring deployment dependencies is adding them into the MANIFEST.MF file:

```
Manifest-Version: 1.0
Dependencies: org.jboss.ws.cxf.jbossws-cxf-client services export,foo.bar
```

Here above *org.jboss.ws.cxf.jbossws-cxf-client* and *foo.bar* are the modules you want to set dependencies to; *services* tells the modules framework that you want to also import *META-INF/services/..* declarations from the dependency, while *export* exports the classes from the module to any other module that might be depending on the module implicitly created for your deployment.

When using annotations on your endpoints / handlers such as the Apache CXF ones (@InInterceptor, @GZIP, ...) remember to add the proper module dependency in your manifest. Otherwise your annotations are not picked up and added to the annotation index by WildFly, resulting in them being completely and silently ignored.

Using JAXB

In order for successfully directly using JAXB contexts, etc. in your client or endpoint running in-container, you need to properly setup a JAXB implementation; that is performed setting the following dependency:

```
Dependencies: com.sun.xml.bind services export
```

Using Apache CXF

In order for using Apache CXF APIs and implementation classes you need to add a dependency to the *org.apache.cxf* (API) module and / or *org.apache.cxf.impl* (implementation) module:

```
Dependencies: org.apache.cxf services
```

However, please note that would not come with any JBossWS-CXF customizations nor additional extensions. For this reason, and generally speaking for simplifying user configuration, a client side aggregation module is available with all the WS dependencies users might need.

Client side WS aggregation module

Whenever you simply want to use all the JBoss Web Services feature/functionalities, you can set a dependency to the convenient client module.

```
Dependencies: org.jboss.ws.cxf.jbossws-cxf-client services
```

Please note the *services* option above: that's strictly required in order for you to get the JBossWS-CXF version of classes that are retrieved using the *Service API*, the `org.apache.cxf.Bus` for instance.

Be careful as issues because of misconfiguration here can be quite hard to track down, because the Apache CXF behaviour would be sensibly different.

The *services* option is almost always needed when declaring dependencies on *org.jboss.ws.cxf.jbossws-cxf-client* and *org.apache.cxf* modules. The reason for this is in it affecting the loading of classes through the *Service API*, which is what is used to wire most of the JBossWS components as well as all Apache CXF Bus extensions.

Annotation scanning

The application server uses an annotation index for detecting JAX-WS endpoints in user deployments. When declaring WS endpoints whose class belongs to a different module (for instance referring that in the `web.xml` descriptor), be sure to have an `annotations` type dependency in place. Without that, your endpoints would simply be ignored as they won't appear as annotated classes to the webservices subsystem.

```
Dependencies: org.foo annotations
```

Using jboss-deployment-descriptor.xml

In some circumstances, the convenient approach of setting module dependencies in MANIFEST.MF might not work. An example is the need for importing/exporting specific resources from a given module dependency. Users should hence add a jboss-deployment-structure.xml descriptor to their deployment and set module dependencies in it.

7 High Availability Guide

- WildFly 8 Cluster Howto
- References
- All WildFly 8 documentation

7.1 Introduction to High Availability Services

7.1.1 What are High Availability services?

WildFly's High Availability services are used to guarantee availability of a deployed Java EE application.

Deploying critical applications on a single node suffers from two potential problems:

- loss of application availability when the node hosting the application crashes (single point of failure)
- loss of application availability in the form of extreme delays in response time during high volumes of requests (overwhelmed server)

WildFly supports two features which ensure high availability of critical Java EE applications:

- **fail-over:** allows a client interacting with a Java EE application to have uninterrupted access to that application, even in the presence of node failures
- **load balancing:** allows a client to have timely responses from the application, even in the presence of high-volumes of requests


These two independent high availability services can very effectively inter-operate when making use of mod_cluster for load balancing!

Taking advantage of WildFly's high availability services is easy, and simply involves deploying WildFly on a cluster of nodes, making a small number of application configuration changes, and then deploying the application in the cluster.

We now take a brief look at what these services can guarantee.

7.1.2 High Availability through fail-over

Fail-over allows a client interacting with a Java EE application to have uninterrupted access to that application, even in the presence of node failures. For example, consider a Java EE application which makes use of the following features:

- session-oriented servlets to provide user interaction
- session-oriented EJBs to perform state-dependent business computation
- EJB entity beans to store critical data in a persistent store (e.g. database)
- SSO login to the application

If the application makes use of WildFly's fail-over services, a client interacting with an instance of that application will not be interrupted even when the node on which that instance executes crashes. Behind the scenes, WildFly makes sure that all of the user data that the application make use of (HTTP session data, EJB SFSB sessions, EJB entities and SSO credentials) are available at other nodes in the cluster, so that when a failure occurs and the client is redirected to that new node for continuation of processing (i.e. the client "fails over" to the new node), the user's data is available and processing can continue.

The Infinispan and JGroups subsystems are instrumental in providing these data availability guarantees and will be discussed in detail later in the guide.

7.1.3 High Availability through load balancing

Load balancing enables the application to respond to client requests in a timely fashion, even when subjected to a high-volume of requests. Using a load balancer as a front-end, each incoming HTTP request can be directed to one node in the cluster for processing. In this way, the cluster acts as a pool of processing nodes and the load is "balanced" over the pool, achieving scalability and, as a consequence, availability. Requests involving session-oriented servlets are directed to the the same application instance in the pool for efficiency of processing (sticky sessions). Using mod_cluster has the advantage that changes in cluster topology (scaling the pool up or down, servers crashing) are communicated back to the load balancer and used to update in real time the load balancing activity and avoid requests being directed to application instances which are no longer available.

The mod_cluster subsystem is instrumental in providing support for this High Availability feature of WildFly and will be discussed in detail later in this guide.

7.1.4 Aims of the guide

This guide aims to:

- provide a description of the high-availability features available in WildFly and the services they depend on
- show how the various high availability services can be configured for particular application use cases
- identify default behavior for features relating to high-availability/clustering

7.1.5 Organization of the guide

As high availability features and their configuration depend on the particular component they affect (e.g. HTTP sessions, EJB SFSB sessions, Hibernate), we organize the discussion around those Java EE features. We strive to make each section as self-contained as possible. Also, when discussing a feature, we will introduce any WildFly subsystems upon which the feature depends.

7.2 HTTP Services

This section summarises the HTTP-based clustering features.

7.2.1 Subsystem Support

This section describes the key clustering subsystems, JGroups and Infinispan. Say a few words about how they work together.

JGroups Subsystem

Purpose

The JGroups subsystem provides group communication support for HA services in the form of JGroups channels.

Named channel instances permit application peers in a cluster to communicate as a group and in such a way that the communication satisfies defined properties (e.g. reliable, ordered, failure-sensitive). Communication properties are configurable for each channel and are defined by the protocol stack used to create the channel. Protocol stacks consist of a base transport layer (used to transport messages around the cluster) together with a user-defined, ordered stack of protocol layers, where each protocol layer supports a given communication property.

The JGroups subsystem provides the following features:

- allows definition of named protocol stacks
- view run-time metrics associated with channels
- specify a default stack for general use

In the following sections, we describe the JGroups subsystem.

JGroups channels are created transparently as part of the clustering functionality (e.g. on clustered application deployment, channels will be created behind the scenes to support clustered features such as session replication or transmission of SSO contexts around the cluster).

Configuration example

What follows is a sample JGroups subsystem configuration showing all of the possible elements and attributes which may be configured. We shall use this example to explain the meaning of the various elements and attributes.

The schema for the subsystem, describing all valid elements and attributes, can be found in the Wildfly distribution, in the docs/schema directory.

```
<subsystem xmlns="urn:jboss:domain:jgroups:5.0">
    <channels default="ee">
        <channel name="ee" stack="udp"/>
    </channels>
    <stacks>
        <stack name="udp">
            <transport type="UDP" socket-binding="jgroups-udp"/>
            <protocol type="PING"/>
            <protocol type="MERGE3"/>
            <protocol type="FD_SOCK"/>
            <protocol type="FD_ALL"/>
            <protocol type="VERIFY_SUSPECT"/>
            <protocol type="pbcast.NAKACK2"/>
            <protocol type="UNICAST3"/>
            <protocol type="pbcast.STABLE"/>
            <protocol type="pbcast.GMS"/>
            <protocol type="UFC"/>
            <protocol type="MFC"/>
            <protocol type="FRAG2"/>
        </stack>
        <stack name="tcp">
            <transport type="TCP" socket-binding="jgroups-tcp"/>
            <socket-protocol type="MPING" socket-binding="jgroups-mping"/>
            <protocol type="MERGE3"/>
            <protocol type="FD_SOCK"/>
            <protocol type="FD_ALL"/>
            <protocol type="VERIFY_SUSPECT"/>
            <protocol type="pbcast.NAKACK2"/>
            <protocol type="UNICAST3"/>
            <protocol type="pbcast.STABLE"/>
            <protocol type="pbcast.GMS"/>
            <protocol type="MFC"/>
            <protocol type="FRAG2"/>
        </stack>
    </stacks>
</subsystem>
```

<subsystem>

This element is used to configure the subsystem within a Wildfly system profile.

- `xmlns` This attribute specifies the XML namespace of the JGroups subsystem and, in particular, its version.
- `default-stack` This attribute is used to specify a default stack for the JGroups subsystem. This default stack will be used whenever a stack is required but no stack is specified.

<stack>

This element is used to configure a JGroups protocol stack.

- `name` This attribute is used to specify the name of the stack.

<transport>

This element is used to configure the transport layer (required) of the protocol stack.

- `type` This attribute specifies the transport type (e.g. UDP, TCP, TCPGOSSIP)
- `socket-binding` This attribute references a defined socket binding in the server profile. It is used when JGroups needs to create general sockets internally.
- `diagnostics-socket-binding` This attribute references a defined socket binding in the server profile. It is used when JGroups needs to create sockets for use with the diagnostics program. For more about the use of diagnostics, see the JGroups documentation for probe.sh.
- `default-executor` This attribute references a defined thread pool executor in the threads subsystem. It governs the allocation and execution of runnable tasks to handle incoming JGroups messages.
- `oob-executor` This attribute references a defined thread pool executor in the threads subsystem. It governs the allocation and execution of runnable tasks to handle incoming JGroups OOB (out-of-bound) messages.
- `timer-executor` This attribute references a defined thread pool executor in the threads subsystem. It governs the allocation and execution of runnable timer-related tasks.
- `shared` This attribute indicates whether or not this transport is shared amongst several JGroups stacks or not.
- `thread-factory` This attribute references a defined thread factory in the threads subsystem. It governs the allocation of threads for running tasks which are not handled by the executors above.
- `site` This attribute defines a site (data centre) id for this node.
- `rack` This attribute defines a rack (server rack) id for this node.
- `machine` This attribute defines a machine (host) is for this node.

site, rack and machine ids are used by the Infinispan topology-aware consistent hash function, which when using dist mode, prevents dist mode replicas from being stored on the same host, rack or site

.

<property>

This element is used to configure a transport property.

- `name` This attribute specifies the name of the protocol property. The value is provided as text for the property element.

<protocol>

This element is used to configure a (non-transport) protocol layer in the JGroups stack. Protocol layers are ordered within the stack.

- `type` This attribute specifies the name of the JGroups protocol implementation (e.g. MPING, pbcast.GMS), with the package prefix org.jgroups.protocols removed.
- `socket-binding` This attribute references a defined socket binding in the server profile. It is used when JGroups needs to create general sockets internally for this protocol instance.

<property>

This element is used to configure a protocol property.

- `name` This attribute specifies the name of the protocol property. The value is provided as text for the property element.

<relay>

This element is used to configure the RELAY protocol for a JGroups stack. RELAY is a protocol which provides cross-site replication between defined sites (data centres). In the RELAY protocol, defined sites specify the names of remote sites (backup sites) to which their data should be backed up. Channels are defined between sites to permit the RELAY protocol to transport the data from the current site to a backup site.

- `site` This attribute specifies the name of the current site. Site names can be referenced elsewhere (e.g. in the JGroups remote-site configuration elements, as well as backup configuration elements in the Infinispan subsystem)

<remote-site>

This element is used to configure a remote site for the RELAY protocol.

- `name` This attribute specifies the name of the remote site to which this configuration applies.
- `stack` This attribute specifies a JGroups protocol stack to use for communication between this site and the remote site.
- `cluster` This attribute specifies the name of the JGroups channel to use for communication between this site and the remote site.

Use Cases

In many cases, channels will be configured via XML as in the example above, so that the channels will be available upon server startup. However, channels may also be added, removed or have their configurations changed in a running server by making use of the Wildfly management API command-line interface (CLI). In this section, we present some key use cases for the JGroups management API.

The key use cases covered are:

- adding a stack
- adding a protocol to an existing stack
- adding a property to a protocol

The Wildfly management API command-line interface (CLI) itself can be used to provide extensive information on the attributes and commands available in the JGroups subsystem interface used in these examples.

Add a stack

```
/subsystem=jgroups/stack=mystack:add()
```

Add a protocol to a stack

```
/subsystem=jgroups/stack=mystack/transport=<type>:add(socket-binding=<socketbinding>)
```

```
/subsystem=jgroups/stack=mystack:protocol=<type>:add(socket-binding=<socketbinding>)
```

Add a property to a protocol

```
/subsystem=jgroups/stack=mystack/transport=<type>:map-put(name=properties, key=<property-name>,
value=<property-value>)
```

Infinispan Subsystem

Purpose

The Infinispan subsystem provides caching support for HA services in the form of Infinispan caches: high-performance, transactional caches which can operate in both non-distributed and distributed scenarios. Distributed caching support is used in the provision of many key HA services. For example, the failover of a session-oriented client HTTP request from a failing node to a new (failover) node depends on session data for the client being available on the new node. In other words, the client session data needs to be replicated across nodes in the cluster. This is effectively achieved via a distributed Infinispan cache. This approach to providing fail-over also applies to EJB SFSB sessions. Over and above providing support for fail-over, an underlying cache is also required when providing second-level caching for entity beans using Hibernate, and this case is also handled through the use of an Infinispan cache.

The Infinispan subsystem provides the following features:

- allows definition and configuration of named cache containers and caches
- view run-time metrics associated with cache container and cache instances

In the following sections, we describe the Infinispan subsystem.

Infiispan cache containers and caches are created transparently as part of the clustering functionality (e.g. on clustered application deployment, cache containers and their associated caches will be created behind the scenes to support clustered features such as session replication or caching of entities around the cluster).

Configuration Example

In this section, we provide an example XML configuration of the infinispan subsystem and review the configuration elements and attributes.

The schema for the subsystem, describing all valid elements and attributes, can be found in the Wildfly distribution, in the docs/schema directory.

```
<subsystem xmlns="urn:jboss:domain:infinispan:4.0">
  <cache-container name="server" aliases="singleton cluster" default-cache="default"
module="org.wildfly.clustering.server">
      <transport lock-timeout="60000"/>
      <replicated-cache name="default" mode="SYNC">
          <transaction mode="BATCH"/>      </replicated-cache>
  </cache-container>
  <cache-container name="web" default-cache="dist"
module="org.wildfly.clustering.web.infinispan">
      <transport lock-timeout="60000"/>
      <replicated-cache name="repl" mode="SYNC">
          <transaction mode="BATCH"/>          <file-store/>
      </replicated-cache>
      <distributed-cache name="dist" mode="SYNC">
          <transaction mode="BATCH"/>
          <file-store/>
      </distributed-cache>
  </cache-container>
  <cache-container name="ejb" aliases="sfsb" default-cache="dist"
module="org.wildfly.clustering.ejb.infinispan">
      <transport lock-timeout="60000"/>
      <replicated-cache name="repl" mode="SYNC">
          <transaction mode="BATCH"/>
          <file-store/>
      </replicated-cache>
      <distributed-cache name="dist" mode="SYNC" l1-lifespan="0">
          <transaction mode="BATCH"/>
          <file-store/>
      </distributed-cache>
  </cache-container>
  <cache-container name="hibernate" module="org.hibernate.infinispan">
      <transport lock-timeout="60000"/>
      <local-cache name="local-query">
          <transaction mode="NONE"/>
          <eviction strategy="LRU" max-entries="10000"/>
          <expiration max-idle="100000"/>
      </local-cache>
      <invalidation-cache name="entity" mode="SYNC">
          <transaction mode="NON_XA"/>
          <eviction strategy="LRU" max-entries="10000"/>
          <expiration max-idle="100000"/>
      </invalidation-cache>
      <replicated-cache name="timestamps" mode="ASYNC">
          <transaction mode="NONE"/>
          <eviction strategy="NONE"/>
       </replicated-cache>
  </cache-container>
</subsystem>
```

<cache-container>

This element is used to configure a cache container.

- `name` This attribute is used to specify the name of the cache container.
- `default-cache` This attribute configures the default cache to be used, when no cache is otherwise specified.
- `listener-executor` This attribute references a defined thread pool executor in the threads subsystem. It governs the allocation and execution of runnable tasks in the replication queue.
- `eviction-executor` This attribute references a defined thread pool executor in the threads subsystem. It governs the allocation and execution of runnable tasks to handle evictions.
- `replication-queue-executor` This attribute references a defined thread pool executor in the threads subsystem. It governs the allocation and execution of runnable tasks to handle asynchronous cache operations.
- `jndi-name` This attribute is used to assign a name for the cache container in the JNDI name service.
- `module` This attribute configures the module whose class loader should be used when building this cache container's configuration.
- `start` This attribute configured the cache container start mode and has since been deprecated, the only supported and the default value is LAZY (on-demand start).
- `aliases` This attribute is used to define aliases for the cache container name.

This element has the following child elements: **<transport>**, **<local-cache>**, **<invalidation-cache>**, **<replicated-cache>**, and **<distributed-cache>**.

<transport>

This element is used to configure the JGroups transport used by the cache container, when required.

- `channel` This attribute configures the JGroups channel to be used for the transport. If none is specified, the default channel as defined by the JGroups subsystem is used.
- `cluster` This attribute configures the name of the group communication cluster. This is the name which will be seen in debugging logs.
- `executor` This attribute references a defined thread pool executor in the threads subsystem. It governs the allocation and execution of runnable tasks to handle ? *<fill me in >*?.
- `lock-timeout` This attribute configures the time-out to be used when obtaining locks for the transport.
- `site` This attribute configures the site id of the cache container.
- `rack` This attribute configures the rack id of the cache container.
- `machine` This attribute configures the machine id of the cache container.

The presence of the transport element is required when operating in clustered mode

The remaining child elements of **<cache-container>**, namely **<local-cache>**, **<invalidation-cache>**, **<replicated-cache>** and **<distributed-cache>**, each configures one of four key cache types or classifications.

These cache-related elements are actually part of an xsd hierarchy with abstract complexTypes **cache**, **clustered-cache**, and **shared-cache**. In order to simplify the presentation, we notate these as pseudo-elements **<abstract cache>**, **<abstract clustered-cache>** and **<abstract shared-cache>**. In what follows, we first describe the extension hierarchy of base elements, and then show how the cache type elements relate to them.

<abstract cache>

This abstract base element defines the attributes and child elements common to all non-clustered caches.

- `name` This attribute configures the name of the cache. This name may be referenced by other subsystems.
- `start` This attribute configured the cache container start mode and has since been deprecated, the only supported and the default value is LAZY (on-demand start).
- `batching` This attribute configures batching. If enabled, the invocation batching API will be made available for this cache.
- `indexing` This attribute configures indexing. If enabled, entries will be indexed when they are added to the cache. Indexes will be updated as entries change or are removed.
- `jndi-name` This attribute is used to assign a name for the cache in the JNDI name service.
- `module` This attribute configures the module whose class loader should be used when building this cache container's configuration.

The <abstract cache> abstract base element has the following child elements: **<indexing-properties>, <locking>**, **<transaction>**, **<eviction>**, **<expiration>**, **<store>**, **<file-store>**, **<string-keyed-jdbc-store>**, **<binary-keyed-jdbc-store>**, **<mixed-keyed-jdbc-store>**, **<remote-store>**.

<indexing-properties>
This child element defines properties to control indexing behaviour.
<locking>
This child element configures the locking behaviour of the cache.

- `isolation` This attribute the cache locking isolation level. Allowable values are NONE, SERIALIZABLE, REPEATABLE_READ, READ_COMMITTED, READ_UNCOMMITTED.
- `striping` If true, a pool of shared locks is maintained for all entries that need to be locked. Otherwise, a lock is created per entry in the cache. Lock striping helps control memory footprint but may reduce concurrency in the system.
- `acquire-timeout` This attribute configures the maximum time to attempt a particular lock acquisition.
- `concurrency-level` This attribute is used to configure the concurrency level. Adjust this value according to the number of concurrent threads interacting with Infinispan.

<transaction>
This child element configures the transactional behaviour of the cache.

- `mode` This attribute configures the transaction mode, setting the cache transaction mode to one of NONE, NON_XA, NON_DURABLE_XA, FULL_XA.
- `stop-timeout` If there are any ongoing transactions when a cache is stopped, Infinispan waits for ongoing remote and local transactions to finish. The amount of time to wait for is defined by the cache stop timeout.
- `locking` This attribute configures the locking mode for this cache, one of OPTIMISTIC or PESSIMISTIC.

<eviction>
This child element configures the eviction behaviour of the cache.

- `strategy` This attribute configures the cache eviction strategy. Available options are 'UNORDERED', 'FIFO', 'LRU', 'LIRS' and 'NONE' (to disable eviction).
- `max-entries` This attribute configures the maximum number of entries in a cache instance. If selected value is not a power of two the actual value will default to the least power of two larger than selected value. -1 means no limit.

<expiration>
This child element configures the expiration behaviour of the cache.

- `max-idle` This attribute configures the maximum idle time a cache entry will be maintained in the cache, in milliseconds. If the idle time is exceeded, the entry will be expired cluster-wide. -1 means the entries never expire.
- `lifespan` This attribute configures the maximum lifespan of a cache entry, after which the entry is expired cluster-wide, in milliseconds. -1 means the entries never expire.
- `interval` This attribute specifies the interval (in ms) between subsequent runs to purge expired entries from memory and any cache stores. If you wish to disable the periodic eviction process altogether, set wakeupInterval to -1.

The remaining child elements of the abstract base element **<cache>**, namely **<store>**, **<file-store>**, **<remote-store>**, **<string-keyed-jdbc-store>**, **<binary-keyed-jdbc-store>** and **<mixed-keyed-jdbc-store>**, each configures one of six key cache store types.

These cache store-related elements are actually part of an xsd extension hierarchy with abstract complexTypes **base-store** and **base-jdbc-store**. As before, in order to simplify the presentation, we notate these as pseudo-elements **<abstract base-store>** and **<abstract base-jdbc-store>**. In what follows, we first describe the extension hierarchy of base elements, and then show how the cache store elements relate to them.

<abstract base-store>
This abstract base element defines the attributes and child elements common to all cache stores.

- `shared` This attribute should be set to true when multiple cache instances share the same cache store (e.g. multiple nodes in a cluster using a JDBC-based CacheStore pointing to the same, shared database) Setting this to true avoids multiple cache instances writing the same modification multiple times. If enabled, only the node where the modification originated will write to the cache store. If disabled, each individual cache reacts to a potential remote update by storing the data to the cache store.
- `preload` This attribute configures whether or not, when the cache starts, data stored in the cache loader will be pre-loaded into memory. This is particularly useful when data in the cache loader is needed immediately after start-up and you want to avoid cache operations being delayed as a result of loading this data lazily. Can be used to provide a 'warm-cache' on start-up, however there is a performance penalty as start-up time is affected by this process. Note that pre-loading is done in a local fashion, so any data loaded is only stored locally in the node. No replication or distribution of the preloaded data happens. Also, Infinispan only pre-loads up to the maximum configured number of entries in eviction.
- `passivation` If true, data is only written to the cache store when it is evicted from memory, a phenomenon known as *passivation*. Next time the data is requested, it will be 'activated' which means that data will be brought back to memory and removed from the persistent store. If false, the cache store contains a copy of the cache contents in memory, so writes to cache result in cache store writes. This essentially gives you a 'write-through' configuration.
- `fetch-state` This attribute, if true, causes persistent state to be fetched when joining a cluster. If multiple cache stores are chained, only one of them can have this property enabled.
- `purge` This attribute configures whether the cache store is purged upon start-up.
- `singleton` This attribute configures whether or not the singleton store cache store is enabled. SingletonStore is a delegating cache store used for situations when only one instance in a cluster should interact with the underlying store.
- `class` This attribute configures a custom store implementation class to use for this cache store.
- `properties` This attribute is used to configure a list of cache store properties.

The abstract base element has one child element: **<write-behind>**
<write-behind>
This element is used to configure a cache store as write-behind instead of write-through. In write-through mode, writes to the cache are also *synchronously* written to the cache store, whereas in write-behind mode, writes to the cache are followed by *asynchronous* writes to the cache store.

- `flush-lock-timeout` This attribute configures the time-out for acquiring the lock which guards the state to be flushed to the cache store periodically.
- `modification-queue-size` This attribute configures the maximum number of entries in the asynchronous queue. When the queue is full, the store becomes write-through until it can accept new entries.
- `shutdown-timeout` This attribute configures the time-out (in ms) to stop the cache store.
- `thread-pool` This attribute is used to configure the size of the thread pool whose threads are responsible for applying the modifications to the cache store.

<abstract base-jdbc-store> extends <abstract base-store>
This abstract base element defines the attributes and child elements common to all JDBC-based cache stores.

- `datasource` This attribute configures the datasource for the JDBC-based cache store.
- `entry-table` This attribute configures the database table used to store cache entries.
- `bucket-table` This attribute configures the database table used to store binary cache entries.

<file-store> extends <abstract base-store>
This child element is used to configure a file-based cache store. This requires specifying the name of the file to be used as backing storage for the cache store.

- `relative-to` This attribute optionally configures a relative path prefix for the file store path. Can be null.
- `path` This attribute configures an absolute path to a file if **relative-to** is null; configures a relative path to the file, in relation to the value for **relative-to**, otherwise.

<remote-store> extends <abstract base-store>
This child element of cache is used to configure a remote cache store. It has a child <remote-servers>.

- `cache` This attribute configures the name of the remote cache to use for this remote store.
- `tcp-nodelay` This attribute configures a TCP_NODELAY value for communication with the remote cache.
- `socket-timeout` This attribute configures a socket time-out for communication with the remote cache.

<remote-servers>
This child element of cache configures a list of remote servers for this cache store.
<remote-server>
This element configures a remote server. A remote server is defined completely by a locally defined outbound socket binding, through which communication is made with the server.

- `outbound-socket-binding` This attribute configures an outbound socket binding for a remote server.

<local-cache> extends <abstract cache>

This element configures a local cache.

<abstract clustered-cache> extends <abstract cache>

This abstract base element defines the attributes and child elements common to all clustered caches. A clustered cache is a cache which spans multiple nodes in a cluster. It inherits from <cache>, so that all attributes and elements of <cache> are also defined for <clustered-cache>.

- `async-marshalling` This attribute configures async marshalling. If enabled, this will cause marshalling of entries to be performed asynchronously.
- `mode` This attribute configures the clustered cache mode, ASYNC for asynchronous operation, or SYNC for synchronous operation.
- `queue-size` In ASYNC mode, this attribute can be used to trigger flushing of the queue when it reaches a specific threshold.
- `queue-flush-interval` In ASYNC mode, this attribute controls how often the asynchronous thread used to flush the replication queue runs. This should be a positive integer which represents thread wakeup time in milliseconds.
- `remote-timeout` In SYNC mode, this attribute (in ms) used to wait for an acknowledgement when making a remote call, after which the call is aborted and an exception is thrown.

<invalidation-cache> extends <abstract clustered-cache>

This element configures an invalidation cache.

<abstract shared-cache> extends <abstract clustered-cache>

This abstract base element defines the attributes and child elements common to all shared caches. A shared cache is a clustered cache which shares state with its cache peers in the cluster. It inherits from <clustered-cache>, so that all attributes and elements of <clustered-cache> are also defined for <shared-cache>.

<state-transfer>

- `enabled` If enabled, this will cause the cache to ask neighbouring caches for state when it starts up, so the cache starts 'warm', although it will impact start-up time.
- `timeout` This attribute configures the maximum amount of time (ms) to wait for state from neighbouring caches, before throwing an exception and aborting start-up.
- `chunk-size` This attribute configures the size, in bytes, in which to batch the transfer of cache entries.

<backups>
<backup>

- `strategy` This attribute configures the backup strategy for this cache. Allowable values are SYNC, ASYNC.
- `failure-policy` This attribute configures the policy to follow when connectivity to the backup site fails. Allowable values are IGNORE, WARN, FAIL, CUSTOM.
- `enabled` This attribute configures whether or not this backup is enabled. If enabled, data will be sent to the backup site; otherwise, the backup site will be effectively ignored.
- `timeout` This attribute configures the time-out for replicating to the backup site.
- `after-failures` This attribute configures the number of failures after which this backup site should go off-line.
- `min-wait` This attribute configures the minimum time (in milliseconds) to wait after the max number of failures is reached, after which this backup site should go off-line.

<backup-for>

- `remote-cache` This attribute configures the name of the remote cache for which this cache acts as a backup.
- `remote-site` This attribute configures the site of the remote cache for which this cache acts as a backup.

<replicated-cache> extends <abstract shared-cache>

This element configures a replicated cache. With a replicated cache, all contents (key-value pairs) of the cache are replicated on all nodes in the cluster.

<distributed-cache> extends <abstract shared-cache>

This element configures a distributed cache. With a distributed cache, contents of the cache are selectively replicated on nodes in the cluster, according to the number of owners specified.

- `owners` This attribute configures the number of cluster-wide replicas for each cache entry.
- `segments` This attribute configures the number of hash space segments which is the granularity for key distribution in the cluster. Value must be strictly positive.
- `l1-lifespan` This attribute configures the maximum lifespan of an entry placed in the L1 cache. Configures the L1 cache behaviour in 'distributed' caches instances. In any other cache modes, this element is ignored.

Use Cases

In many cases, cache containers and caches will be configured via XML as in the example above, so that they will be available upon server start-up. However, cache containers and caches may also be added, removed or have their configurations changed in a running server by making use of the Wildfly management API command-line interface (CLI). In this section, we present some key use cases for the Infinispan management API.

The key use cases covered are:

- adding a cache container
- adding a cache to an existing cache container
- configuring the transaction subsystem of a cache

The Wildfly management API command-line interface (CLI) can be used to provide extensive information on the attributes and commands available in the Infinispan subsystem interface used in these examples.

Add a cache container

```
/subsystem=infinispan/cache-container=mycontainer:add(default-cache=<default-cache-name>)
/subsystem=infinispan/cache-container=mycontainer/transport=jgroups:add(lock-timeout=<timeout>)
```

Add a cache

```
/subsystem=infinispan/cache-container=mycontainer/local-cache=mylocalcache:add()
```

Configure the transaction component of a cache

```
/subsystem=infinispan/cache-container=mycontainer/local-cache=mylocalcache/component=transaction:a
```

7.2.2 Clustered Web Sessions

7.2.3 Clustered SSO

7.2.4 Load Balancing

This section describes load balancing via Apache + mod_jk and Apache + mod_cluster.

7.2.5 Load balancing with Apache + mod_jk

Describe load balancing with Apache using mod_jk.

7.2.6 Load balancing with Apache + mod_cluster

Describe load balancing with Apache using mod_cluster.

mod_cluster Subsystem

The mod_cluster integration is done via the modcluster subsystem.

7.3 Configuration

7.3.1 Instance ID or JVMRoute

The instance-id or JVMRoute defaults to jboss.node.name property passed on server startup (e.g. via -Djboss.node.name=XYZ).

```
[standalone@localhost:9990 /] /subsystem=undertow/:read-attribute(name=instance-id)
{
    "outcome" => "success",
    "result" => expression "${jboss.node.name}"
}
```

To configure instance-id statically, configure the corresponding property in Undertow subsystem:

```
[standalone@localhost:9990 /]
/subsystem=undertow/:write-attribute(name=instance-id,value=myroute)
{
    "outcome" => "success",
    "response-headers" => {
        "operation-requires-reload" => true,
        "process-state" => "reload-required"
    }
}
```

7.3.2 Proxies

By default, mod_cluster is configured for multicast-based discovery. To specify a static list of proxies, create a remote-socket-binding for each proxy and then reference them in the 'proxies' attribute. See the following example for configuration in the domain mode:

```
[domain@localhost:9990 /]
/socket-binding-group=ha-sockets/remote-destination-outbound-socket-binding=proxy1:add(host=10.21.
port=6666)
{
    "outcome" => "success",
    "result" => undefined,
    "server-groups" => undefined
}
[domain@localhost:9990 /]
/socket-binding-group=ha-sockets/remote-destination-outbound-socket-binding=proxy2:add(host=10.21.
port=6666)
{
    "outcome" => "success",
    "result" => undefined,
    "server-groups" => undefined
}
[domain@localhost:9990 /]
/profile=ha/subsystem=modcluster/mod-cluster-config=configuration/:write-attribute(name=proxies,
value=[proxy1, proxy2]
{
    "outcome" => "success",
    "result" => undefined,
    "server-groups" => undefined
}
[domain@localhost:9990 /] :reload-servers
{
    "outcome" => "success",
    "result" => undefined,
    "server-groups" => undefined
}
```

7.4 Runtime Operations

The modcluster subsystem supports several operations:

```
[standalone@localhost:9999 subsystem=modcluster] :read-operation-names
{
    "outcome" => "success",
    "result" => [
        "add",
        "add-custom-metric",
        "add-metric",
        "add-proxy",
        "disable",
        "disable-context",
        "enable",
        "enable-context",
        "list-proxies",
        "read-attribute",
        "read-children-names",
        "read-children-resources",
        "read-children-types",
        "read-operation-description",
        "read-operation-names",
        "read-proxies-configuration",
        "read-proxies-info",
        "read-resource",
        "read-resource-description",
        "refresh",
        "remove-custom-metric",
        "remove-metric",
        "remove-proxy",
        "reset",
        "stop",
        "stop-context",
        "validate-address",
        "write-attribute"
    ]
}
```

The operations specific to the modcluster subsystem are divided in 3 categories the ones that affects the configuration and require a restart of the subsystem, the one that just modify the behaviour temporarily and the ones that display information from the httpd part.

7.4.1 operations displaying httpd informations

There are 2 operations that display how Apache httpd sees the node:

read-proxies-configuration

Send a DUMP message to all Apache httpd the node is connected to and display the message received from Apache httpd.

```
[standalone@localhost:9999 subsystem=modcluster] :read-proxies-configuration
{
    "outcome" => "success",
    "result" => [
        "neo3:6666",
        "balancer: [1] Name: mycluster Sticky: 1 [JSESSIONID]/[jsessionid] remove: 0 force: 1
Timeout: 0 Maxtry: 1
node: [1:1],Balancer: mycluster,JVMRoute: 498bb1f0-00d9-3436-a341-7f012bc2e7ec,Domain: [],Host:
127.0.0.1,Port: 8080,Type: http,flushpackets: 0,flushwait: 10,ping: 10,smax: 26,ttl: 60,timeout:
0
host: 1 [example.com] vhost: 1 node: 1
host: 2 [localhost] vhost: 1 node: 1
host: 3 [default-host] vhost: 1 node: 1
context: 1 [/myapp] vhost: 1 node: 1 status: 1
context: 2 [/] vhost: 1 node: 1 status: 1
",
        "jfcpc:6666",
        "balancer: [1] Name: mycluster Sticky: 1 [JSESSIONID]/[jsessionid] remove: 0 force: 1
Timeout: 0 maxAttempts: 1
node: [1:1],Balancer: mycluster,JVMRoute: 498bb1f0-00d9-3436-a341-7f012bc2e7ec,LBGroup: [],Host:
127.0.0.1,Port: 8080,Type: http,flushpackets: 0,flushwait: 10,ping: 10,smax: 26,ttl: 60,timeout:
0
host: 1 [default-host] vhost: 1 node: 1
host: 2 [localhost] vhost: 1 node: 1
host: 3 [example.com] vhost: 1 node: 1
context: 1 [/] vhost: 1 node: 1 status: 1
context: 2 [/myapp] vhost: 1 node: 1 status: 1
"
    ]
}
```

read-proxies-info

Send a INFO message to all Apache httpd the node is connected to and display the message received from Apache httpd.

```
[standalone@localhost:9999 subsystem=modcluster] :read-proxies-info
{
    "outcome" => "success",
    "result" => [
        "neo3:6666",
        "Node: [1],Name: 498bb1f0-00d9-3436-a341-7f012bc2e7ec,Balancer: mycluster,Domain: ,Host:
127.0.0.1,Port: 8080,Type: http,Flushpackets: Off,Flushwait: 10000,Ping: 10000000,Smax: 26,Ttl:
60000000,Elected: 0,Read: 0,Transfered: 0,Connected: 0,Load: -1
Vhost: [1:1:1], Alias: example.com
Vhost: [1:1:2], Alias: localhost
Vhost: [1:1:3], Alias: default-host
Context: [1:1:1], Context: /myapp, Status: ENABLED
Context: [1:1:2], Context: /, Status: ENABLED
",
        "jfcpc:6666",
        "Node: [1],Name: 498bb1f0-00d9-3436-a341-7f012bc2e7ec,Balancer: mycluster,LBGroup:
,Host: 127.0.0.1,Port: 8080,Type: http,Flushpackets: Off,Flushwait: 10,Ping: 10,Smax: 26,Ttl:
60,Elected: 0,Read: 0,Transfered: 0,Connected: 0,Load: 1
Vhost: [1:1:1], Alias: default-host
Vhost: [1:1:2], Alias: localhost
Vhost: [1:1:3], Alias: example.com
Context: [1:1:1], Context: /, Status: ENABLED
Context: [1:1:2], Context: /myapp, Status: ENABLED
"
    ]
}
```

7.4.2

operations that handle the proxies the node is connected too

there are 3 operation that could be used to manipulate the list of Apache httpd the node is connected too.

list-proxies:

Displays the httpd that are connected to the node. The httpd could be discovered via the Advertise protocol or via the proxy-list attribute.

```
[standalone@localhost:9999 subsystem=modcluster] :list-proxies
{
    "outcome" => "success",
    "result" => [
        "proxy1:6666",
        "proxy2:6666"
    ]
}
```

remove-proxy

Remove a proxy from the discovered proxies or temporarily from the proxy-list attribute.

```
[standalone@localhost:9999 subsystem=modcluster] :remove-proxy(host=jfcpc, port=6666)
{"outcome" => "success"}
```

proxy

Add a proxy to the discovered proxies or temporarily to the proxy-list attribute.

```
[standalone@localhost:9999 subsystem=modcluster] :add-proxy(host=jfcpc, port=6666)
{"outcome" => "success"}
```

7.4.3 Context related operations

Those operations allow to send context related commands to Apache httpd. They are send automatically when deploying or undeploying webapps.

enable-context

Tell Apache httpd that the context is ready receive requests.

```
[standalone@localhost:9999 subsystem=modcluster] :enable-context(context=/myapp,
virtualhost=default-host)
{"outcome" => "success"}
```

disable-context

Tell Apache httpd that it shouldn't send new session requests to the context of the virtualhost.

```
[standalone@localhost:9999 subsystem=modcluster] :disable-context(context=/myapp,
virtualhost=default-host)
{"outcome" => "success"}
```

stop-context

Tell Apache httpd that it shouldn't send requests to the context of the virtualhost.

```
[standalone@localhost:9999 subsystem=modcluster] :stop-context(context=/myapp,
virtualhost=default-host, waittime=50)
{"outcome" => "success"}
```

7.4.4 Node related operations

Those operations are like the context operation but they apply to all webapps running on the node and operation that affect the whole node.

refresh

Refresh the node by sending a new CONFIG message to Apache httpd.

reset

reset the connection between Apache httpd and the node

7.4.5 Configuration

Metric configuration

There are 4 metric operations corresponding to add and remove load metrics to the dynamic-load-provider. Note that when nothing is defined a simple-load-provider is use with a fixed load factor of one.

```
[standalone@localhost:9999 subsystem=modcluster] :read-resource(name=mod-cluster-config)
{
    "outcome" => "success",
    "result" => {"simple-load-provider" => {"factor" => "1"}}
}
```

that corresponds to the following configuration:

```
<subsystem xmlns="urn:jboss:domain:modcluster:1.0">
            <mod-cluster-config>
                <simple-load-provider factor="1"/>
            </mod-cluster-config>
 </subsystem>
```

metric

Add a metric to the dynamic-load-provider, the dynamic-load-provider in configuration is created if needed.

```
[standalone@localhost:9999 subsystem=modcluster] :add-metric(type=cpu)
{"outcome" => "success"}
[standalone@localhost:9999 subsystem=modcluster] :read-resource(name=mod-cluster-config)
{
    "outcome" => "success",
    "result" => {
        "dynamic-load-provider" => {
            "history" => 9,
            "decay" => 2,
            "load-metric" => [{
                "type" => "cpu"
            }]
        }
    }
}
```

remove-metric

Remove a metric from the dynamic-load-provider.

```
[standalone@localhost:9999 subsystem=modcluster] :remove-metric(type=cpu)
{"outcome" => "success"}
```

custom-metric / remove-custom-metric

like the add-metric and remove-metric except they require a class parameter instead the type. Usually they needed additional properties which can be specified

```
[standalone@localhost:9999 subsystem=modcluster] :add-custom-metric(class=myclass,
property=[("pro1" => "value1"), ("pro2" => "value2")]
{"outcome" => "success"}
```

which corresponds the following in the xml configuration file:

```
<subsystem xmlns="urn:jboss:domain:modcluster:1.0">
            <mod-cluster-config>
                <dynamic-load-provider history="9" decay="2">
                    <custom-load-metric class="myclass">
                        <property name="pro1" value="value1"/>
                        <property name="pro2" value="value2"/>
                    </custom-load-metric>
                </dynamic-load-provider>
            </mod-cluster-config>
</subsystem>
```

7.5 EJB Services

This chapter explains how clustering of EJBs works in WildFly 8.

7.5.1 EJB Subsystem

7.6 EJB Timer

Wildfly now supports clustered database backed timers. For details have a look to the EJB3 reference section

7.6.1 Marking an EJB as clustered

WildFly 8 allows clustering of stateful session beans. A stateful session bean can be marked with `@org.jboss.ejb3.annotation.Clustered` annotation or be marked as clustered using the jboss-ejb3.xml's `<clustered>` element.

MyStatefulBean

```
import org.jboss.ejb3.annotation.Clustered;
import javax.ejb.Stateful;

@Stateful
@Clustered
public class MyStatefulBean {
...
}
```

jboss-ejb3.xml

```
<jboss xmlns="http://www.jboss.com/xml/ns/javaee"
       xmlns:jee="http://java.sun.com/xml/ns/javaee"
       xmlns:c="urn:clustering:1.0">


    <jee:assembly-descriptor>
        <c:clustering>
            <jee:ejb-name>DDBasedClusteredBean</jee:ejb-name>
            <c:clustered>true</c:clustered>
        </c:clustering>
    </jee:assembly-descriptor>
</jboss>
```

7.6.2 Deploying clustered EJBs

Clustering support is available in the HA profiles of WildFly 8. In this chapter we'll be using the standalone server for explaining the details. However, the same applies to servers in a domain mode. Starting the standalone server with HA capabilities enabled, involves starting it with the standalone-ha.xml (or even standalone-full-ha.xml):

```
./standalone.sh -server-config=standalone-ha.xml
```

This will start a single instance of the server with HA capabilities. Deploying the EJBs to this instance *doesn't* involve anything special and is the same as explained in the application deployment chapter.

Obviously, to be able to see the benefits of clustering, you'll need more than one instance of the server. So let's start another server with HA capabilities. That another instance of the server can either be on the same machine or on some other machine. If it's on the same machine, the two things you have to make sure is that you pass the port offset for the second instance and also make sure that each of the server instances have a unique `jboss.node.name` system property. You can do that by passing the following two system properties to the startup command:

```
./standalone.sh -server-config=standalone-ha.xml -Djboss.socket.binding.port-offset=<offset of
your choice> -Djboss.node.name=<unique node name>
```

Follow whichever approach you feel comfortable with for deploying the EJB deployment to this instance too.

Deploying the application on just one node of a standalone instance of a clustered server does **not** mean that it will be automatically deployed to the other clustered instance. You will have to do deploy it explicitly on the other standalone clustered instance too. Or you can start the servers in domain mode so that the deployment can be deployed to all the server within a server group. See the admin guide for more details on domain setup.

Now that you have deployed an application with clustered EJBs on both the instances, the EJBs are now capable of making use of the clustering features.

7.6.3 Failover for clustered EJBs

Clustered EJBs have failover capability. The state of the @Stateful @Clustered EJBs is replicated across the cluster nodes so that if one of the nodes in the cluster goes down, some other node will be able to take over the invocations. Let's see how it's implemented in WildFly 8. In the next few sections we'll see how it works for remote (standalone) clients and for clients in another remote WildFly server instance. Although, there isn't a difference in how it works in both these cases, we'll still explain it separately so as to make sure there aren't any unanswered questions.

Remote standalone clients

In this section we'll consider a remote standalone client (i.e. a client which runs in a separate JVM and *isn't* running within another WildFly 8 instance). Let's consider that we have 2 servers, server X and server Y which we started earlier. Each of these servers has the clustered EJB deployment. A standalone remote client can use either the JNDI approach or native JBoss EJB client APIs to communicate with the servers. The important thing to note is that when you are invoking clustered EJB deployments, you do **not** have to list all the servers within the cluster (which obviously wouldn't have been feasible due the dynamic nature of cluster node additions within a cluster).

The remote client just has to list only one of the servers with the clustering capability. In this case, we can either list server X (in jboss-ejb-client.properties) *or* server Y. This server will act as the starting point for cluster topology communication between the client and the clustered nodes.

Note that you have to configure the *ejb* cluster in the jboss-ejb-client.properties configuration file, like so:

```
remote.clusters=ejb
remote.cluster.ejb.connect.options.org.xnio.Options.SASL_POLICY_NOANONYMOUS=false
remote.cluster.ejb.connect.options.org.xnio.Options.SSL_ENABLED=false
```

Cluster topology communication

When a client connects to a server, the JBoss EJB client implementation (internally) communicates with the server for cluster topology information, if the server had clustering capability. In our example above, let's assume we listed server X as the initial server to connect to. When the client connects to server X, the server will send back an (asynchronous) cluster topology message to the client. This topology message consists of the cluster name(s) and the information of the nodes that belong to the cluster. The node information includes the node address and port number to connect to (whenever necessary). So in this example, the server X will send back the cluster topology consisting of the other server Y which belongs to the cluster.

In case of stateful (clustered) EJBs, a typical invocation flow involves creating of a session for the stateful bean, which happens when you do a JNDI lookup for that bean, and then invoking on the returned proxy. The lookup for stateful bean, internally, triggers a (synchronous) session creation request from the client to the server. In this case, the session creation request goes to server X since that's the initial connection that we have configured in our jboss-ejb-client.properties. Since server X is clustered, it will return back a session id and along with send back an *"affinity"* of that session. In case of clustered servers, the affinity equals to the name of the cluster to which the stateful bean belongs on the server side. For non-clustered beans, the affinity is just the node name on which the session was created. This *affinity* will later help the EJB client to route the invocations on the proxy, appropriately to either a node within a cluster (for clustered beans) or to a specific node (for non-clustered beans). While this session creation request is going on, the server X will also send back an asynchronous message which contains the cluster topology. The JBoss EJB client implementation will take note of this topology information and will later use it for connection creation to nodes within the cluster and routing invocations to those nodes, whenever necessary.

Now that we know how the cluster topology information is communicated from the server to the client, let see how failover works. Let's continue with the example of server X being our starting point and a client application looking up a stateful bean and invoking on it. During these invocations, the client side will have collected the cluster topology information from the server. Now let's assume for some reason, server X goes down and the client application subsequent invokes on the proxy. The JBoss EJB client implementation, at this stage will be aware of the affinity and in this case it's a cluster affinity. Because of the cluster topology information it has, it knows that the cluster has two nodes server X and server Y. When the invocation now arrives, it sees that the server X is down. So it uses a selector to fetch a suitable node from among the cluster nodes. The selector itself is configurable, but we'll leave it from discussion for now. When the selector returns a node from among the cluster, the JBoss EJB client implementation creates a connection to that node (if not already created earlier) and creates a EJB receiver out of it. Since in our example, the only other node in the cluster is server Y, the selector will return that node and the JBoss EJB client implementation will use it to create a EJB receiver out of it and use that receiver to pass on the invocation on the proxy. Effectively, the invocation has now failed over to a different node within the cluster.

Remote clients on another instance of WildFly 8

So far we discussed remote standalone clients which typically use either the EJB client API or the jboss-ejb-client.properties based approach to configure and communicate with the servers where the clustered beans are deployed. Now let's consider the case where the client is an application deployed another AS7 instance and it wants to invoke on a clustered stateful bean which is deployed on another instance of WildFly 8. In this example let's consider a case where we have 3 servers involved. Server X and Server Y both belong to a cluster and have clustered EJB deployed on them. Let's consider another server instance Server C (which may or may *not* have clustering capability) which acts as a client on which there's a deployment which wants to invoke on the clustered beans deployed on server X and Y and achieve failover.

The configurations required to achieve this are explained in this chapter. As you can see the configurations are done in a jboss-ejb-client.xml which points to a remote outbound connection to the other server. This jboss-ejb-client.xml goes in the deployment of server C (since that's our client). As explained eariler, the client configuration need **not** point to all clustered nodes. Instead it just has to point to one of them which will act as a start point for communication. So in this case, we can create a remote outbound connection on server C to server X and use server X as our starting point for communication. Just like in the case of remote standalone clients, when the application on server C (client) looks up a stateful bean, a session creation request will be sent to server X which will send back a session id and the cluster affinity for it. Furthermore, server X asynchronously send back a message to server C (client) containing the cluster topology. This topology information will include the node information of server Y (since that belongs to the cluster along with server X). Subsequent invocations on the proxy will be routed appropriately to the nodes in the cluster. If server X goes down, as explained earlier, a different node from the cluster will be selected and the invocation will be forwarded to that node.

As can be seen both remote standalone client and remote clients on another WildFly 8 instance act similar in terms of failover.

Testcases for failover of stateful beans

We have testcases in WildFly 8 testsuite which test that whatever is explained above works as expected. The RemoteEJBClientStatefulBeanFailoverTestCase tests the case where a stateful EJB uses @Clustered annotation to mark itself as clustered. We also have RemoteEJBClientDDBasedSFSBFailoverTestCase which uses jboss-ejb3.xml to mark a stateful EJB as clustered. Both these testcases test that when a node goes down in a cluster, the client invocation is routed to a different node in the cluster.

7.7 Hibernate

7.8 HA Singleton Features

In general, an HA or clustered singleton is a service that exists on multiple nodes in a cluster, but is active on just a single node at any given time. If the node providing the service fails or is shut down, a new singleton provider is chosen and started. Thus, other than a brief interval when one provider has stopped and another has yet to start, the service is always running on one node.

7.8.1 Singleton subsystem

WildFly 10 introduces a “singleton” subsystem, which defines a set of policies that define how an HA singleton should behave. A singleton policy can be used to instrument singleton deployments or to create singleton MSC services.

Configuration

The default subsystem configuration from WildFly’s ha and full-ha profile looks like:

```
<subsystem xmlns="urn:jboss:domain:singleton:1.0">
    <singleton-policies default="default">
        <singleton-policy name="default" cache-container="server">
            <simple-election-policy/>
        </singleton-policy>
    </singleton-policies>
</subsystem>
```

A singleton policy defines:

1. A unique name
2. A cache container and cache with which to register singleton provider candidates
3. An election policy
4. A quorum (optional)

One can add a new singleton policy via the following management operation:

```
/subsystem=singleton/singleton-policy=foo:add(cache-container=server)
```

Cache configuration

The cache-container and cache attributes of a singleton policy must reference a valid cache from the Infinispan subsystem. If no specific cache is defined, the default cache of the cache container is assumed. This cache is used as a registry of which nodes can provide a given service and will typically use a replicated-cache configuration.

Election policies

WildFly 10 includes 2 singleton election policy implementations:

- **simple**
  Elects the provider (a.k.a. master) of a singleton service based on a specified position in a circular linked list of eligible nodes sorted by descending age. Position=0, the default value, refers to the oldest node, 1 is second oldest, etc. ; while position=-1 refers to the youngest node, -2 to the second youngest, etc.
  e.g.

```
/subsystem=singleton/singleton-policy=foo/election-policy=simple:add(position=-1)
```

- **random**
  Elects a random member to be the provider of a singleton service
  e.g.

```
/subsystem=singleton/singleton-policy=foo/election-policy=random:add()
```

Preferences

Additionally, any singleton election policy may indicate a preference for one or more members of a cluster. Preferences may be defined either via node name or via outbound socket binding name. Node preferences always take precedent over the results of an election policy.
e.g.

```
/subsystem=singleton/singleton-policy=foo/election-policy=simple:list-add(name=name-preferences,
value=nodeA)
/subsystem=singleton/singleton-policy=bar/election-policy=random:list-add(name=socket-binding-pref
value=nodeA)
```

Quorum

Network partitions are particularly problematic for singleton services, since they can trigger multiple singleton providers for the same service to run at the same time. To defend against this scenario, a singleton policy may define a quorum that requires a minimum number of nodes to be present before a singleton provider election can take place. A typical deployment scenario uses a quorum of N/2 + 1, where N is the anticipated cluster size. This value can be updated at runtime, and will immediately affect any active singleton services.
e.g.

```
/subsystem=singleton/singleton-policy=foo:write-attribute(name=quorum, value=3)
```

HA environments

The singleton subsystem can be used in a non-HA profile, so long as the cache that it references uses a local-cache configuration. In this manner, an application leveraging singleton functionality (via the singleton API or using a singleton deployment descriptor) will continue function as if the server was a sole member of a cluster. For obvious reasons, the use of a quorum does not make sense in such a configuration.

7.8.2 Singleton deployments

WildFly 10 resurrects the ability to start a given deployment on a single node in the cluster at any given time. If that node shuts down, or fails, the application will automatically start on another node on which the given deployment exists. Long time users of JBoss AS will recognize this functionality as being akin to the HASingletonDeployer, a.k.a. “deploy-hasingleton”, feature of AS6 and earlier.

Usage

A deployment indicates that it should be deployed as a singleton via a deployment descriptor. This can either be a standalone “/META-INF/singleton-deployment.xml” file or embedded within an existing jboss-all.xml descriptor. This descriptor may be applied to any deployment type, e.g. JAR, WAR, EAR, etc., with the exception of a subdeployment within an EAR.
e.g.

```
<singleton-deployment xmlns="urn:jboss:singleton-deployment:1.0" policy="foo"/>
```

The singleton deployment descriptor defines which singleton policy should be used to deploy the application. If undefined, the default singleton policy is used, as defined by the singleton subsystem.

Using a standalone descriptor is often preferable, since it may be overlaid onto an existing deployment archive.
e.g.

```
deployment-overlay add --name=singleton-policy-foo
--content=/META-INF/singleton-deployment.xml=/path/to/singleton-deployment.xml
--deployments=my-app.jar --redeploy-affected
```

7.8.3 Singleton MSC services

WildFly allows any user MSC service to be installed as a singleton MSC service via a public API. Once installed, the service will only ever start on 1 node in the cluster at a time. If the node providing the service is shutdown, or fails, another node on which the service was installed will start automatically.

Installing an MSC service using an existing singleton policy

While singleton MSC services have been around since AS7, WildFly 10 adds the ability to leverage the singleton subsystem to create singleton MSC services from existing singleton policies.

The singleton subsystem exposes capabilities for each singleton policy it defines. These policies, represented via the `org.wildfly.clustering.singleton.SingletonPolicy` interface, can be referenced via the following name: “org.wildfly.clustering.singleton.policy”
e.g.

```
public class MyServiceActivator implements ServiceActivator {
    @Override
    public void activate(ServiceActivatorContext context) {
        ServiceName name = ServiceName.parse(“my.service.name”);
        Service<?> service = new MyService();
        try {
            SingletonPolicy policy = (SingletonPolicy)
context.getServiceRegistry().getRequiredService(ServiceName.parse(SingletonPolicy.CAPABILITY_NAME)
policy.createSingletonServiceBuilder(name, service).build(context.getServiceTarget()).install();
        } catch (InterruptedException e) {
            throw new ServiceRegistryException(e);
        }
    }
}
```

Installing an MSC service using dynamic singleton policy

Alternatively, you can build singleton policy dynamically, which is particularly useful if you want to use a custom singleton election policy. Specifically, `SingletonPolicy` is a generalization of the `org.wildfly.clustering.singleton.SingletonServiceBuilderFactory` interface, which includes support for specifying an election policy and, optionally, a quorum.
e.g.

```
public class MyServiceActivator implements ServiceActivator {
    @Override
    public void activate(ServiceActivatorContext context) {
        String containerName = “server”;
        ElectionPolicy policy = new MySingletonElectionPolicy();
        int quorum = 3;
        ServiceName name = ServiceName.parse(“my.service.name”);
        Service<?> service = new MyService();
        try {
            SingletonServiceBuilderFactory factory = (SingletonServiceBuilderFactory)
context.getServiceRegistry().getRequiredService(SingletonServiceName.BUILDER.getServiceName(contai
factory.createSingletonServiceBuilder(name, service)
                .electionPolicy(policy)
                .quorum(quorum)
                .build(context.getServiceTarget()).install();
        } catch (InterruptedException e) {
            throw new ServiceRegistryException(e);
        }
    }
}
```

7.9 Related Issues

Couldn't find a page to include called: Related Issues

7.10 Changes From Previous Versions

Describe here key changes between releases.

7.10.1 Key changes

7.10.2 Migration to Wildfly

7.11 WildFly 8 Cluster Howto

Couldn't find a page to include called: WildFly 8 Cluster Howto

7.12 References

Couldn't find a page to include called: References

7.13 All WildFly 8 documentation

Couldn't find a page to include called: All WildFly 8 documentation

7.14 Introduction To High Availability Services

7.14.1 What are High Availability services?

WildFly's High Availability services are used to guarantee availability of a deployed Java EE application.

Deploying critical applications on a single node suffers from two potential problems:

- loss of application availability when the node hosting the application crashes (single point of failure)
- loss of application availability in the form of extreme delays in response time during high volumes of requests (overwhelmed server)

WildFly supports two features which ensure high availability of critical Java EE applications:

- **fail-over:** allows a client interacting with a Java EE application to have uninterrupted access to that application, even in the presence of node failures
- **load balancing:** allows a client to have timely responses from the application, even in the presence of high-volumes of requests

These two independent high availability services can very effectively inter-operate when making use of mod_cluster for load balancing!

Taking advantage of WildFly's high availability services is easy, and simply involves deploying WildFly on a cluster of nodes, making a small number of application configuration changes, and then deploying the application in the cluster.

We now take a brief look at what these services can guarantee.

7.14.2 High Availability through fail-over

Fail-over allows a client interacting with a Java EE application to have uninterrupted access to that application, even in the presence of node failures. For example, consider a Java EE application which makes use of the following features:

- session-oriented servlets to provide user interaction
- session-oriented EJBs to perform state-dependent business computation
- EJB entity beans to store critical data in a persistent store (e.g. database)
- SSO login to the application

If the application makes use of WildFly's fail-over services, a client interacting with an instance of that application will not be interrupted even when the node on which that instance executes crashes. Behind the scenes, WildFly makes sure that all of the user data that the application make use of (HTTP session data, EJB SFSB sessions, EJB entities and SSO credentials) are available at other nodes in the cluster, so that when a failure occurs and the client is redirected to that new node for continuation of processing (i.e. the client "fails over" to the new node), the user's data is available and processing can continue.

The Infinispan and JGroups subsystems are instrumental in providing these data availability guarantees and will be discussed in detail later in the guide.

7.14.3 High Availability through load balancing

Load balancing enables the application to respond to client requests in a timely fashion, even when subjected to a high-volume of requests. Using a load balancer as a front-end, each incoming HTTP request can be directed to one node in the cluster for processing. In this way, the cluster acts as a pool of processing nodes and the load is "balanced" over the pool, achieving scalability and, as a consequence, availability. Requests involving session-oriented servlets are directed to the the same application instance in the pool for efficiency of processing (sticky sessions). Using mod_cluster has the advantage that changes in cluster topology (scaling the pool up or down, servers crashing) are communicated back to the load balancer and used to update in real time the load balancing activity and avoid requests being directed to application instances which are no longer available.

The mod_cluster subsystem is instrumental in providing support for this High Availability feature of WildFly and will be discussed in detail later in this guide.

7.14.4 Aims of the guide

This guide aims to:

- provide a description of the high-availability features available in WildFly and the services they depend on
- show how the various high availability services can be configured for particular application use cases
- identify default behavior for features relating to high-availability/clustering

7.14.5 Organization of the guide

As high availability features and their configuration depend on the particular component they affect (e.g. HTTP sessions, EJB SFSB sessions, Hibernate), we organize the discussion around those Java EE features. We strive to make each section as self-contained as possible. Also, when discussing a feature, we will introduce any WildFly subsystems upon which the feature depends.

7.15 Subsystem Support

This section describes the key clustering subsystems, JGroups and Infinispan. Say a few words about how they work together.

7.15.1 JGroups Subsystem

7.15.2 Purpose

The JGroups subsystem provides group communication support for HA services in the form of JGroups channels.

Named channel instances permit application peers in a cluster to communicate as a group and in such a way that the communication satisfies defined properties (e.g. reliable, ordered, failure-sensitive). Communication properties are configurable for each channel and are defined by the protocol stack used to create the channel. Protocol stacks consist of a base transport layer (used to transport messages around the cluster) together with a user-defined, ordered stack of protocol layers, where each protocol layer supports a given communication property.

The JGroups subsystem provides the following features:

- allows definition of named protocol stacks
- view run-time metrics associated with channels
- specify a default stack for general use

In the following sections, we describe the JGroups subsystem.

JGroups channels are created transparently as part of the clustering functionality (e.g. on clustered application deployment, channels will be created behind the scenes to support clustered features such as session replication or transmission of SSO contexts around the cluster).

7.15.3 Configuration example

What follows is a sample JGroups subsystem configuration showing all of the possible elements and attributes which may be configured. We shall use this example to explain the meaning of the various elements and attributes.

The schema for the subsystem, describing all valid elements and attributes, can be found in the Wildfly distribution, in the docs/schema directory.

```
<subsystem xmlns="urn:jboss:domain:jgroups:5.0">
    <channels default="ee">
        <channel name="ee" stack="udp"/>
    </channels>
    <stacks>
        <stack name="udp">
            <transport type="UDP" socket-binding="jgroups-udp"/>
            <protocol type="PING"/>
            <protocol type="MERGE3"/>
            <protocol type="FD_SOCK"/>
            <protocol type="FD_ALL"/>
            <protocol type="VERIFY_SUSPECT"/>
            <protocol type="pbcast.NAKACK2"/>
            <protocol type="UNICAST3"/>
            <protocol type="pbcast.STABLE"/>
            <protocol type="pbcast.GMS"/>
            <protocol type="UFC"/>
            <protocol type="MFC"/>
            <protocol type="FRAG2"/>
        </stack>
        <stack name="tcp">
            <transport type="TCP" socket-binding="jgroups-tcp"/>
            <socket-protocol type="MPING" socket-binding="jgroups-mping"/>
            <protocol type="MERGE3"/>
            <protocol type="FD_SOCK"/>
            <protocol type="FD_ALL"/>
            <protocol type="VERIFY_SUSPECT"/>
            <protocol type="pbcast.NAKACK2"/>
            <protocol type="UNICAST3"/>
            <protocol type="pbcast.STABLE"/>
            <protocol type="pbcast.GMS"/>
            <protocol type="MFC"/>
            <protocol type="FRAG2"/>
        </stack>
    </stacks>
</subsystem>
```

<subsystem>

This element is used to configure the subsystem within a Wildfly system profile.

- `xmlns` This attribute specifies the XML namespace of the JGroups subsystem and, in particular, its version.
- `default-stack` This attribute is used to specify a default stack for the JGroups subsystem. This default stack will be used whenever a stack is required but no stack is specified.

<stack>

This element is used to configure a JGroups protocol stack.

- `name` This attribute is used to specify the name of the stack.

<transport>

This element is used to configure the transport layer (required) of the protocol stack.

- `type` This attribute specifies the transport type (e.g. UDP, TCP, TCPGOSSIP)
- `socket-binding` This attribute references a defined socket binding in the server profile. It is used when JGroups needs to create general sockets internally.
- `diagnostics-socket-binding` This attribute references a defined socket binding in the server profile. It is used when JGroups needs to create sockets for use with the diagnostics program. For more about the use of diagnostics, see the JGroups documentation for probe.sh.
- `default-executor` This attribute references a defined thread pool executor in the threads subsystem. It governs the allocation and execution of runnable tasks to handle incoming JGroups messages.
- `oob-executor` This attribute references a defined thread pool executor in the threads subsystem. It governs the allocation and execution of runnable tasks to handle incoming JGroups OOB (out-of-bound) messages.
- `timer-executor` This attribute references a defined thread pool executor in the threads subsystem. It governs the allocation and execution of runnable timer-related tasks.
- `shared` This attribute indicates whether or not this transport is shared amongst several JGroups stacks or not.
- `thread-factory` This attribute references a defined thread factory in the threads subsystem. It governs the allocation of threads for running tasks which are not handled by the executors above.
- `site` This attribute defines a site (data centre) id for this node.
- `rack` This attribute defines a rack (server rack) id for this node.
- `machine` This attribute defines a machine (host) is for this node.

site, rack and machine ids are used by the Infinispan topology-aware consistent hash function, which when using dist mode, prevents dist mode replicas from being stored on the same host, rack or site

.

<property>

This element is used to configure a transport property.

- `name` This attribute specifies the name of the protocol property. The value is provided as text for the property element.

<protocol>

This element is used to configure a (non-transport) protocol layer in the JGroups stack. Protocol layers are ordered within the stack.

- `type` This attribute specifies the name of the JGroups protocol implementation (e.g. MPING, pbcast.GMS), with the package prefix org.jgroups.protocols removed.
- `socket-binding` This attribute references a defined socket binding in the server profile. It is used when JGroups needs to create general sockets internally for this protocol instance.

<property>

This element is used to configure a protocol property.

- `name` This attribute specifies the name of the protocol property. The value is provided as text for the property element.

<relay>

This element is used to configure the RELAY protocol for a JGroups stack. RELAY is a protocol which provides cross-site replication between defined sites (data centres). In the RELAY protocol, defined sites specify the names of remote sites (backup sites) to which their data should be backed up. Channels are defined between sites to permit the RELAY protocol to transport the data from the current site to a backup site.

- `site` This attribute specifies the name of the current site. Site names can be referenced elsewhere (e.g. in the JGroups remote-site configuration elements, as well as backup configuration elements in the Infinispan subsystem)

<remote-site>

This element is used to configure a remote site for the RELAY protocol.

- `name` This attribute specifies the name of the remote site to which this configuration applies.
- `stack` This attribute specifies a JGroups protocol stack to use for communication between this site and the remote site.
- `cluster` This attribute specifies the name of the JGroups channel to use for communication between this site and the remote site.

7.15.4 Use Cases

In many cases, channels will be configured via XML as in the example above, so that the channels will be available upon server startup. However, channels may also be added, removed or have their configurations changed in a running server by making use of the Wildfly management API command-line interface (CLI). In this section, we present some key use cases for the JGroups management API.

The key use cases covered are:

- adding a stack
- adding a protocol to an existing stack
- adding a property to a protocol

The Wildfly management API command-line interface (CLI) itself can be used to provide extensive information on the attributes and commands available in the JGroups subsystem interface used in these examples.

Add a stack

```
/subsystem=jgroups/stack=mystack:add()
```

Add a protocol to a stack

```
/subsystem=jgroups/stack=mystack/transport=<type>:add(socket-binding=<socketbinding>)
```

```
/subsystem=jgroups/stack=mystack:protocol=<type>:add(socket-binding=<socketbinding>)
```

Add a property to a protocol

```
/subsystem=jgroups/stack=mystack/transport=<type>:map-put(name=properties, key=<property-name>,
value=<property-value>)
```

Infinispan Subsystem

7.15.5 Purpose

The Infinispan subsystem provides caching support for HA services in the form of Infinispan caches: high-performance, transactional caches which can operate in both non-distributed and distributed scenarios. Distributed caching support is used in the provision of many key HA services. For example, the failover of a session-oriented client HTTP request from a failing node to a new (failover) node depends on session data for the client being available on the new node. In other words, the client session data needs to be replicated across nodes in the cluster. This is effectively achieved via a distributed Infinispan cache. This approach to providing fail-over also applies to EJB SFSB sessions. Over and above providing support for fail-over, an underlying cache is also required when providing second-level caching for entity beans using Hibernate, and this case is also handled through the use of an Infinispan cache.

The Infinispan subsystem provides the following features:

- allows definition and configuration of named cache containers and caches
- view run-time metrics associated with cache container and cache instances

In the following sections, we describe the Infinispan subsystem.

Infiispan cache containers and caches are created transparently as part of the clustering functionality (e.g. on clustered application deployment, cache containers and their associated caches will be created behind the scenes to support clustered features such as session replication or caching of entities around the cluster).

7.15.6 Configuration Example

In this section, we provide an example XML configuration of the infinispan subsystem and review the configuration elements and attributes.

The schema for the subsystem, describing all valid elements and attributes, can be found in the Wildfly distribution, in the docs/schema directory.

```
<subsystem xmlns="urn:jboss:domain:infinispan:4.0">
  <cache-container name="server" aliases="singleton cluster" default-cache="default"
module="org.wildfly.clustering.server">
      <transport lock-timeout="60000"/>
      <replicated-cache name="default" mode="SYNC">
          <transaction mode="BATCH"/>      </replicated-cache>
  </cache-container>
  <cache-container name="web" default-cache="dist"
module="org.wildfly.clustering.web.infinispan">
      <transport lock-timeout="60000"/>
      <replicated-cache name="repl" mode="SYNC">
          <transaction mode="BATCH"/>          <file-store/>
      </replicated-cache>
      <distributed-cache name="dist" mode="SYNC">
          <transaction mode="BATCH"/>
          <file-store/>
      </distributed-cache>
  </cache-container>
  <cache-container name="ejb" aliases="sfsb" default-cache="dist"
module="org.wildfly.clustering.ejb.infinispan">
      <transport lock-timeout="60000"/>
      <replicated-cache name="repl" mode="SYNC">
          <transaction mode="BATCH"/>
          <file-store/>
      </replicated-cache>
      <distributed-cache name="dist" mode="SYNC" l1-lifespan="0">
          <transaction mode="BATCH"/>
          <file-store/>
      </distributed-cache>
  </cache-container>
  <cache-container name="hibernate" module="org.hibernate.infinispan">
      <transport lock-timeout="60000"/>
      <local-cache name="local-query">
          <transaction mode="NONE"/>
          <eviction strategy="LRU" max-entries="10000"/>
          <expiration max-idle="100000"/>
      </local-cache>
      <invalidation-cache name="entity" mode="SYNC">
          <transaction mode="NON_XA"/>
          <eviction strategy="LRU" max-entries="10000"/>
          <expiration max-idle="100000"/>
      </invalidation-cache>
      <replicated-cache name="timestamps" mode="ASYNC">
          <transaction mode="NONE"/>
          <eviction strategy="NONE"/>
       </replicated-cache>
  </cache-container>
</subsystem>
```

<cache-container>

This element is used to configure a cache container.

- `name` This attribute is used to specify the name of the cache container.
- `default-cache` This attribute configures the default cache to be used, when no cache is otherwise specified.
- `listener-executor` This attribute references a defined thread pool executor in the threads subsystem. It governs the allocation and execution of runnable tasks in the replication queue.
- `eviction-executor` This attribute references a defined thread pool executor in the threads subsystem. It governs the allocation and execution of runnable tasks to handle evictions.
- `replication-queue-executor` This attribute references a defined thread pool executor in the threads subsystem. It governs the allocation and execution of runnable tasks to handle asynchronous cache operations.
- `jndi-name` This attribute is used to assign a name for the cache container in the JNDI name service.
- `module` This attribute configures the module whose class loader should be used when building this cache container's configuration.
- `start` This attribute configured the cache container start mode and has since been deprecated, the only supported and the default value is LAZY (on-demand start).
- `aliases` This attribute is used to define aliases for the cache container name.

This element has the following child elements: **<transport>**, **<local-cache>**, **<invalidation-cache>**, **<replicated-cache>**, and **<distributed-cache>**.

<transport>

This element is used to configure the JGroups transport used by the cache container, when required.

- `channel` This attribute configures the JGroups channel to be used for the transport. If none is specified, the default channel as defined by the JGroups subsystem is used.
- `cluster` This attribute configures the name of the group communication cluster. This is the name which will be seen in debugging logs.
- `executor` This attribute references a defined thread pool executor in the threads subsystem. It governs the allocation and execution of runnable tasks to handle ? *<fill me in >*?.
- `lock-timeout` This attribute configures the time-out to be used when obtaining locks for the transport.
- `site` This attribute configures the site id of the cache container.
- `rack` This attribute configures the rack id of the cache container.
- `machine` This attribute configures the machine id of the cache container.

The presence of the transport element is required when operating in clustered mode

The remaining child elements of **<cache-container>**, namely **<local-cache>**, **<invalidation-cache>**, **<replicated-cache>** and **<distributed-cache>**, each configures one of four key cache types or classifications.

These cache-related elements are actually part of an xsd hierarchy with abstract complexTypes **cache**, **clustered-cache**, and **shared-cache**. In order to simplify the presentation, we notate these as pseudo-elements **<abstract cache>**, **<abstract clustered-cache>** and **<abstract shared-cache>**. In what follows, we first describe the extension hierarchy of base elements, and then show how the cache type elements relate to them.

<abstract cache>

This abstract base element defines the attributes and child elements common to all non-clustered caches.

- `name` This attribute configures the name of the cache. This name may be referenced by other subsystems.
- `start` This attribute configured the cache container start mode and has since been deprecated, the only supported and the default value is LAZY (on-demand start).
- `batching` This attribute configures batching. If enabled, the invocation batching API will be made available for this cache.
- `indexing` This attribute configures indexing. If enabled, entries will be indexed when they are added to the cache. Indexes will be updated as entries change or are removed.
- `jndi-name` This attribute is used to assign a name for the cache in the JNDI name service.
- `module` This attribute configures the module whose class loader should be used when building this cache container's configuration.

The <abstract cache> abstract base element has the following child elements: **<indexing-properties>, <locking>**, **<transaction>**, **<eviction>**, **<expiration>**, **<store>**, **<file-store>**, **<string-keyed-jdbc-store>**, **<binary-keyed-jdbc-store>**, **<mixed-keyed-jdbc-store>**, **<remote-store>**.

<indexing-properties>

This child element defines properties to control indexing behaviour.

<locking>

This child element configures the locking behaviour of the cache.

- `isolation` This attribute the cache locking isolation level. Allowable values are NONE, SERIALIZABLE, REPEATABLE_READ, READ_COMMITTED, READ_UNCOMMITTED.
- `striping` If true, a pool of shared locks is maintained for all entries that need to be locked. Otherwise, a lock is created per entry in the cache. Lock striping helps control memory footprint but may reduce concurrency in the system.
- `acquire-timeout` This attribute configures the maximum time to attempt a particular lock acquisition.
- `concurrency-level` This attribute is used to configure the concurrency level. Adjust this value according to the number of concurrent threads interacting with Infinispan.

<transaction>

This child element configures the transactional behaviour of the cache.

- `mode` This attribute configures the transaction mode, setting the cache transaction mode to one of NONE, NON_XA, NON_DURABLE_XA, FULL_XA.
- `stop-timeout` If there are any ongoing transactions when a cache is stopped, Infinispan waits for ongoing remote and local transactions to finish. The amount of time to wait for is defined by the cache stop timeout.
- `locking` This attribute configures the locking mode for this cache, one of OPTIMISTIC or PESSIMISTIC.

<eviction>

This child element configures the eviction behaviour of the cache.

- `strategy` This attribute configures the cache eviction strategy. Available options are 'UNORDERED', 'FIFO', 'LRU', 'LIRS' and 'NONE' (to disable eviction).
- `max-entries` This attribute configures the maximum number of entries in a cache instance. If selected value is not a power of two the actual value will default to the least power of two larger than selected value. -1 means no limit.

<expiration>

This child element configures the expiration behaviour of the cache.

- `max-idle` This attribute configures the maximum idle time a cache entry will be maintained in the cache, in milliseconds. If the idle time is exceeded, the entry will be expired cluster-wide. -1 means the entries never expire.
- `lifespan` This attribute configures the maximum lifespan of a cache entry, after which the entry is expired cluster-wide, in milliseconds. -1 means the entries never expire.
- `interval` This attribute specifies the interval (in ms) between subsequent runs to purge expired entries from memory and any cache stores. If you wish to disable the periodic eviction process altogether, set wakeupInterval to -1.

The remaining child elements of the abstract base element **<cache>**, namely **<store>**, **<file-store>**, **<remote-store>**, **<string-keyed-jdbc-store>**, **<binary-keyed-jdbc-store>** and **<mixed-keyed-jdbc-store>**, each configures one of six key cache store types.

These cache store-related elements are actually part of an xsd extension hierarchy with abstract complexTypes **base-store** and **base-jdbc-store**. As before, in order to simplify the presentation, we notate these as pseudo-elements **<abstract base-store>** and **<abstract base-jdbc-store>**. In what follows, we first describe the extension hierarchy of base elements, and then show how the cache store elements relate to them.

<abstract base-store>

This abstract base element defines the attributes and child elements common to all cache stores.

- `shared` This attribute should be set to true when multiple cache instances share the same cache store (e.g. multiple nodes in a cluster using a JDBC-based CacheStore pointing to the same, shared database) Setting this to true avoids multiple cache instances writing the same modification multiple times. If enabled, only the node where the modification originated will write to the cache store. If disabled, each individual cache reacts to a potential remote update by storing the data to the cache store.
- `preload` This attribute configures whether or not, when the cache starts, data stored in the cache loader will be pre-loaded into memory. This is particularly useful when data in the cache loader is needed immediately after start-up and you want to avoid cache operations being delayed as a result of loading this data lazily. Can be used to provide a 'warm-cache' on start-up, however there is a performance penalty as start-up time is affected by this process. Note that pre-loading is done in a local fashion, so any data loaded is only stored locally in the node. No replication or distribution of the preloaded data happens. Also, Infinispan only pre-loads up to the maximum configured number of entries in eviction.
- `passivation` If true, data is only written to the cache store when it is evicted from memory, a phenomenon known as *passivation*. Next time the data is requested, it will be 'activated' which means that data will be brought back to memory and removed from the persistent store. If false, the cache store contains a copy of the cache contents in memory, so writes to cache result in cache store writes. This essentially gives you a 'write-through' configuration.
- `fetch-state` This attribute, if true, causes persistent state to be fetched when joining a cluster. If multiple cache stores are chained, only one of them can have this property enabled.
- `purge` This attribute configures whether the cache store is purged upon start-up.
- `singleton` This attribute configures whether or not the singleton store cache store is enabled. SingletonStore is a delegating cache store used for situations when only one instance in a cluster should interact with the underlying store.
- `class` This attribute configures a custom store implementation class to use for this cache store.
- `properties` This attribute is used to configure a list of cache store properties.

The abstract base element has one child element: **<write-behind>**

<write-behind>

This element is used to configure a cache store as write-behind instead of write-through. In write-through mode, writes to the cache are also *synchronously* written to the cache store, whereas in write-behind mode, writes to the cache are followed by *asynchronous* writes to the cache store.

- `flush-lock-timeout` This attribute configures the time-out for acquiring the lock which guards the state to be flushed to the cache store periodically.
- `modification-queue-size` This attribute configures the maximum number of entries in the asynchronous queue. When the queue is full, the store becomes write-through until it can accept new entries.
- `shutdown-timeout` This attribute configures the time-out (in ms) to stop the cache store.
- `thread-pool` This attribute is used to configure the size of the thread pool whose threads are responsible for applying the modifications to the cache store.

<abstract base-jdbc-store> extends <abstract base-store>

This abstract base element defines the attributes and child elements common to all JDBC-based cache stores.

- `datasource` This attribute configures the datasource for the JDBC-based cache store.
- `entry-table` This attribute configures the database table used to store cache entries.
- `bucket-table` This attribute configures the database table used to store binary cache entries.

<file-store> extends <abstract base-store>

This child element is used to configure a file-based cache store. This requires specifying the name of the file to be used as backing storage for the cache store.

- `relative-to` This attribute optionally configures a relative path prefix for the file store path. Can be null.
- `path` This attribute configures an absolute path to a file if **relative-to** is null; configures a relative path to the file, in relation to the value for **relative-to**, otherwise.

<remote-store> extends <abstract base-store>

This child element of cache is used to configure a remote cache store. It has a child <remote-servers>.

- `cache` This attribute configures the name of the remote cache to use for this remote store.
- `tcp-nodelay` This attribute configures a TCP_NODELAY value for communication with the remote cache.
- `socket-timeout` This attribute configures a socket time-out for communication with the remote cache.

<remote-servers>

This child element of cache configures a list of remote servers for this cache store.

<remote-server>

This element configures a remote server. A remote server is defined completely by a locally defined outbound socket binding, through which communication is made with the server.

- `outbound-socket-binding` This attribute configures an outbound socket binding for a remote server.

<local-cache> extends <abstract cache>

This element configures a local cache.

<abstract clustered-cache> extends <abstract cache>

This abstract base element defines the attributes and child elements common to all clustered caches. A clustered cache is a cache which spans multiple nodes in a cluster. It inherits from <cache>, so that all attributes and elements of <cache> are also defined for <clustered-cache>.

- `async-marshalling` This attribute configures async marshalling. If enabled, this will cause marshalling of entries to be performed asynchronously.
- `mode` This attribute configures the clustered cache mode, ASYNC for asynchronous operation, or SYNC for synchronous operation.
- `queue-size` In ASYNC mode, this attribute can be used to trigger flushing of the queue when it reaches a specific threshold.
- `queue-flush-interval` In ASYNC mode, this attribute controls how often the asynchronous thread used to flush the replication queue runs. This should be a positive integer which represents thread wakeup time in milliseconds.
- `remote-timeout` In SYNC mode, this attribute (in ms) used to wait for an acknowledgement when making a remote call, after which the call is aborted and an exception is thrown.

<invalidation-cache> extends <abstract clustered-cache>

This element configures an invalidation cache.

<abstract shared-cache> extends <abstract clustered-cache>

This abstract base element defines the attributes and child elements common to all shared caches. A shared cache is a clustered cache which shares state with its cache peers in the cluster. It inherits from <clustered-cache>, so that all attributes and elements of <clustered-cache> are also defined for <shared-cache>.

<state-transfer>

- `enabled` If enabled, this will cause the cache to ask neighbouring caches for state when it starts up, so the cache starts 'warm', although it will impact start-up time.
- `timeout` This attribute configures the maximum amount of time (ms) to wait for state from neighbouring caches, before throwing an exception and aborting start-up.
- `chunk-size` This attribute configures the size, in bytes, in which to batch the transfer of cache entries.

<backups>

<backup>

- `strategy` This attribute configures the backup strategy for this cache. Allowable values are SYNC, ASYNC.
- `failure-policy` This attribute configures the policy to follow when connectivity to the backup site fails. Allowable values are IGNORE, WARN, FAIL, CUSTOM.
- `enabled` This attribute configures whether or not this backup is enabled. If enabled, data will be sent to the backup site; otherwise, the backup site will be effectively ignored.
- `timeout` This attribute configures the time-out for replicating to the backup site.
- `after-failures` This attribute configures the number of failures after which this backup site should go off-line.
- `min-wait` This attribute configures the minimum time (in milliseconds) to wait after the max number of failures is reached, after which this backup site should go off-line.

<backup-for>

- `remote-cache` This attribute configures the name of the remote cache for which this cache acts as a backup.
- `remote-site` This attribute configures the site of the remote cache for which this cache acts as a backup.

<replicated-cache> extends <abstract shared-cache>

This element configures a replicated cache. With a replicated cache, all contents (key-value pairs) of the cache are replicated on all nodes in the cluster.

<distributed-cache> extends <abstract shared-cache>

This element configures a distributed cache. With a distributed cache, contents of the cache are selectively replicated on nodes in the cluster, according to the number of owners specified.

- `owners` This attribute configures the number of cluster-wide replicas for each cache entry.
- `segments` This attribute configures the number of hash space segments which is the granularity for key distribution in the cluster. Value must be strictly positive.
- `l1-lifespan` This attribute configures the maximum lifespan of an entry placed in the L1 cache. Configures the L1 cache behaviour in 'distributed' caches instances. In any other cache modes, this element is ignored.

7.15.7 Use Cases

In many cases, cache containers and caches will be configured via XML as in the example above, so that they will be available upon server start-up. However, cache containers and caches may also be added, removed or have their configurations changed in a running server by making use of the Wildfly management API command-line interface (CLI). In this section, we present some key use cases for the Infinispan management API.

The key use cases covered are:

- adding a cache container
- adding a cache to an existing cache container
- configuring the transaction subsystem of a cache

> ⚠ The Wildfly management API command-line interface (CLI) can be used to provide extensive information on the attributes and commands available in the Infinispan subsystem interface used in these examples.

Add a cache container

```
/subsystem=infinispan/cache-container=mycontainer:add(default-cache=<default-cache-name>)
/subsystem=infinispan/cache-container=mycontainer/transport=jgroups:add(lock-timeout=<timeout>)
```

Add a cache

```
/subsystem=infinispan/cache-container=mycontainer/local-cache=mylocalcache:add()
```

Configure the transaction component of a cache

```
/subsystem=infinispan/cache-container=mycontainer/local-cache=mylocalcache/component=transaction:a
```

7.15.8 JGroups Subsystem

Purpose

The JGroups subsystem provides group communication support for HA services in the form of JGroups channels.

Named channel instances permit application peers in a cluster to communicate as a group and in such a way that the communication satisfies defined properties (e.g. reliable, ordered, failure-sensitive). Communication properties are configurable for each channel and are defined by the protocol stack used to create the channel. Protocol stacks consist of a base transport layer (used to transport messages around the cluster) together with a user-defined, ordered stack of protocol layers, where each protocol layer supports a given communication property.

The JGroups subsystem provides the following features:

- allows definition of named protocol stacks
- view run-time metrics associated with channels
- specify a default stack for general use

In the following sections, we describe the JGroups subsystem.

JGroups channels are created transparently as part of the clustering functionality (e.g. on clustered application deployment, channels will be created behind the scenes to support clustered features such as session replication or transmission of SSO contexts around the cluster).

Configuration example

What follows is a sample JGroups subsystem configuration showing all of the possible elements and attributes which may be configured. We shall use this example to explain the meaning of the various elements and attributes.

The schema for the subsystem, describing all valid elements and attributes, can be found in the Wildfly distribution, in the docs/schema directory.

```
<subsystem xmlns="urn:jboss:domain:jgroups:5.0">
    <channels default="ee">
        <channel name="ee" stack="udp"/>
    </channels>
    <stacks>
        <stack name="udp">
            <transport type="UDP" socket-binding="jgroups-udp"/>
            <protocol type="PING"/>
            <protocol type="MERGE3"/>
            <protocol type="FD_SOCK"/>
            <protocol type="FD_ALL"/>
            <protocol type="VERIFY_SUSPECT"/>
            <protocol type="pbcast.NAKACK2"/>
            <protocol type="UNICAST3"/>
            <protocol type="pbcast.STABLE"/>
            <protocol type="pbcast.GMS"/>
            <protocol type="UFC"/>
            <protocol type="MFC"/>
            <protocol type="FRAG2"/>
        </stack>
        <stack name="tcp">
            <transport type="TCP" socket-binding="jgroups-tcp"/>
            <socket-protocol type="MPING" socket-binding="jgroups-mping"/>
            <protocol type="MERGE3"/>
            <protocol type="FD_SOCK"/>
            <protocol type="FD_ALL"/>
            <protocol type="VERIFY_SUSPECT"/>
            <protocol type="pbcast.NAKACK2"/>
            <protocol type="UNICAST3"/>
            <protocol type="pbcast.STABLE"/>
            <protocol type="pbcast.GMS"/>
            <protocol type="MFC"/>
            <protocol type="FRAG2"/>
        </stack>
    </stacks>
</subsystem>
```

<subsystem>

This element is used to configure the subsystem within a Wildfly system profile.

- `xmlns` This attribute specifies the XML namespace of the JGroups subsystem and, in particular, its version.
- `default-stack` This attribute is used to specify a default stack for the JGroups subsystem. This default stack will be used whenever a stack is required but no stack is specified.

<stack>

This element is used to configure a JGroups protocol stack.

- `name` This attribute is used to specify the name of the stack.

<transport>

This element is used to configure the transport layer (required) of the protocol stack.

- `type` This attribute specifies the transport type (e.g. UDP, TCP, TCPGOSSIP)
- `socket-binding` This attribute references a defined socket binding in the server profile. It is used when JGroups needs to create general sockets internally.
- `diagnostics-socket-binding` This attribute references a defined socket binding in the server profile. It is used when JGroups needs to create sockets for use with the diagnostics program. For more about the use of diagnostics, see the JGroups documentation for probe.sh.
- `default-executor` This attribute references a defined thread pool executor in the threads subsystem. It governs the allocation and execution of runnable tasks to handle incoming JGroups messages.
- `oob-executor` This attribute references a defined thread pool executor in the threads subsystem. It governs the allocation and execution of runnable tasks to handle incoming JGroups OOB (out-of-bound) messages.
- `timer-executor` This attribute references a defined thread pool executor in the threads subsystem. It governs the allocation and execution of runnable timer-related tasks.
- `shared` This attribute indicates whether or not this transport is shared amongst several JGroups stacks or not.
- `thread-factory` This attribute references a defined thread factory in the threads subsystem. It governs the allocation of threads for running tasks which are not handled by the executors above.
- `site` This attribute defines a site (data centre) id for this node.
- `rack` This attribute defines a rack (server rack) id for this node.
- `machine` This attribute defines a machine (host) is for this node.

site, rack and machine ids are used by the Infinispan topology-aware consistent hash function, which when using dist mode, prevents dist mode replicas from being stored on the same host, rack or site

.

<property>

This element is used to configure a transport property.

- `name` This attribute specifies the name of the protocol property. The value is provided as text for the property element.

<protocol>

This element is used to configure a (non-transport) protocol layer in the JGroups stack. Protocol layers are ordered within the stack.

- `type` This attribute specifies the name of the JGroups protocol implementation (e.g. MPING, pbcast.GMS), with the package prefix org.jgroups.protocols removed.
- `socket-binding` This attribute references a defined socket binding in the server profile. It is used when JGroups needs to create general sockets internally for this protocol instance.

<property>

This element is used to configure a protocol property.

- `name` This attribute specifies the name of the protocol property. The value is provided as text for the property element.

<relay>

This element is used to configure the RELAY protocol for a JGroups stack. RELAY is a protocol which provides cross-site replication between defined sites (data centres). In the RELAY protocol, defined sites specify the names of remote sites (backup sites) to which their data should be backed up. Channels are defined between sites to permit the RELAY protocol to transport the data from the current site to a backup site.

- `site` This attribute specifies the name of the current site. Site names can be referenced elsewhere (e.g. in the JGroups remote-site configuration elements, as well as backup configuration elements in the Infinispan subsystem)

<remote-site>

This element is used to configure a remote site for the RELAY protocol.

- `name` This attribute specifies the name of the remote site to which this configuration applies.
- `stack` This attribute specifies a JGroups protocol stack to use for communication between this site and the remote site.
- `cluster` This attribute specifies the name of the JGroups channel to use for communication between this site and the remote site.

Use Cases

In many cases, channels will be configured via XML as in the example above, so that the channels will be available upon server startup. However, channels may also be added, removed or have their configurations changed in a running server by making use of the Wildfly management API command-line interface (CLI). In this section, we present some key use cases for the JGroups management API.

The key use cases covered are:

- adding a stack
- adding a protocol to an existing stack
- adding a property to a protocol

The Wildfly management API command-line interface (CLI) itself can be used to provide extensive information on the attributes and commands available in the JGroups subsystem interface used in these examples.

Add a stack

```
/subsystem=jgroups/stack=mystack:add()
```

Add a protocol to a stack

```
/subsystem=jgroups/stack=mystack/transport=<type>:add(socket-binding=<socketbinding>)
```

```
/subsystem=jgroups/stack=mystack:protocol=<type>:add(socket-binding=<socketbinding>)
```

Add a property to a protocol

```
/subsystem=jgroups/stack=mystack/transport=<type>:map-put(name=properties, key=<property-name>,
value=<property-value>)
```

7.15.9 Infinispan Subsystem

Purpose

The Infinispan subsystem provides caching support for HA services in the form of Infinispan caches: high-performance, transactional caches which can operate in both non-distributed and distributed scenarios. Distributed caching support is used in the provision of many key HA services. For example, the failover of a session-oriented client HTTP request from a failing node to a new (failover) node depends on session data for the client being available on the new node. In other words, the client session data needs to be replicated across nodes in the cluster. This is effectively achieved via a distributed Infinispan cache. This approach to providing fail-over also applies to EJB SFSB sessions. Over and above providing support for fail-over, an underlying cache is also required when providing second-level caching for entity beans using Hibernate, and this case is also handled through the use of an Infinispan cache.

The Infinispan subsystem provides the following features:

- allows definition and configuration of named cache containers and caches
- view run-time metrics associated with cache container and cache instances

In the following sections, we describe the Infinispan subsystem.

Infiispan cache containers and caches are created transparently as part of the clustering functionality (e.g. on clustered application deployment, cache containers and their associated caches will be created behind the scenes to support clustered features such as session replication or caching of entities around the cluster).

Configuration Example

In this section, we provide an example XML configuration of the infinispan subsystem and review the configuration elements and attributes.

The schema for the subsystem, describing all valid elements and attributes, can be found in the Wildfly distribution, in the docs/schema directory.

```
<subsystem xmlns="urn:jboss:domain:infinispan:4.0">
  <cache-container name="server" aliases="singleton cluster" default-cache="default"
module="org.wildfly.clustering.server">
      <transport lock-timeout="60000"/>
      <replicated-cache name="default" mode="SYNC">
          <transaction mode="BATCH"/>      </replicated-cache>
  </cache-container>
  <cache-container name="web" default-cache="dist"
module="org.wildfly.clustering.web.infinispan">
      <transport lock-timeout="60000"/>
      <replicated-cache name="repl" mode="SYNC">
          <transaction mode="BATCH"/>          <file-store/>
      </replicated-cache>
      <distributed-cache name="dist" mode="SYNC">
          <transaction mode="BATCH"/>
          <file-store/>
      </distributed-cache>
  </cache-container>
  <cache-container name="ejb" aliases="sfsb" default-cache="dist"
module="org.wildfly.clustering.ejb.infinispan">
      <transport lock-timeout="60000"/>
      <replicated-cache name="repl" mode="SYNC">
          <transaction mode="BATCH"/>
          <file-store/>
      </replicated-cache>
      <distributed-cache name="dist" mode="SYNC" l1-lifespan="0">
          <transaction mode="BATCH"/>
          <file-store/>
      </distributed-cache>
  </cache-container>
  <cache-container name="hibernate" module="org.hibernate.infinispan">
      <transport lock-timeout="60000"/>
      <local-cache name="local-query">
          <transaction mode="NONE"/>
          <eviction strategy="LRU" max-entries="10000"/>
          <expiration max-idle="100000"/>
      </local-cache>
      <invalidation-cache name="entity" mode="SYNC">
          <transaction mode="NON_XA"/>
          <eviction strategy="LRU" max-entries="10000"/>
          <expiration max-idle="100000"/>
      </invalidation-cache>
      <replicated-cache name="timestamps" mode="ASYNC">
          <transaction mode="NONE"/>
          <eviction strategy="NONE"/>
       </replicated-cache>
  </cache-container>
</subsystem>
```

<cache-container>

This element is used to configure a cache container.

- `name` This attribute is used to specify the name of the cache container.
- `default-cache` This attribute configures the default cache to be used, when no cache is otherwise specified.
- `listener-executor` This attribute references a defined thread pool executor in the threads subsystem. It governs the allocation and execution of runnable tasks in the replication queue.
- `eviction-executor` This attribute references a defined thread pool executor in the threads subsystem. It governs the allocation and execution of runnable tasks to handle evictions.
- `replication-queue-executor` This attribute references a defined thread pool executor in the threads subsystem. It governs the allocation and execution of runnable tasks to handle asynchronous cache operations.
- `jndi-name` This attribute is used to assign a name for the cache container in the JNDI name service.
- `module` This attribute configures the module whose class loader should be used when building this cache container's configuration.
- `start` This attribute configured the cache container start mode and has since been deprecated, the only supported and the default value is LAZY (on-demand start).
- `aliases` This attribute is used to define aliases for the cache container name.

This element has the following child elements: **<transport>**, **<local-cache>**, **<invalidation-cache>**, **<replicated-cache>**, and **<distributed-cache>**.

<transport>

This element is used to configure the JGroups transport used by the cache container, when required.

- `channel` This attribute configures the JGroups channel to be used for the transport. If none is specified, the default channel as defined by the JGroups subsystem is used.
- `cluster` This attribute configures the name of the group communication cluster. This is the name which will be seen in debugging logs.
- `executor` This attribute references a defined thread pool executor in the threads subsystem. It governs the allocation and execution of runnable tasks to handle ? *<fill me in >*?.
- `lock-timeout` This attribute configures the time-out to be used when obtaining locks for the transport.
- `site` This attribute configures the site id of the cache container.
- `rack` This attribute configures the rack id of the cache container.
- `machine` This attribute configures the machine id of the cache container.

The presence of the transport element is required when operating in clustered mode

The remaining child elements of **<cache-container>**, namely **<local-cache>**, **<invalidation-cache>**, **<replicated-cache>** and **<distributed-cache>**, each configures one of four key cache types or classifications.

These cache-related elements are actually part of an xsd hierarchy with abstract complexTypes **cache**, **clustered-cache**, and **shared-cache**. In order to simplify the presentation, we notate these as pseudo-elements **<abstract cache>**, **<abstract clustered-cache>** and **<abstract shared-cache>**. In what follows, we first describe the extension hierarchy of base elements, and then show how the cache type elements relate to them.

<abstract cache>

This abstract base element defines the attributes and child elements common to all non-clustered caches.

- `name` This attribute configures the name of the cache. This name may be referenced by other subsystems.
- `start` This attribute configured the cache container start mode and has since been deprecated, the only supported and the default value is LAZY (on-demand start).
- `batching` This attribute configures batching. If enabled, the invocation batching API will be made available for this cache.
- `indexing` This attribute configures indexing. If enabled, entries will be indexed when they are added to the cache. Indexes will be updated as entries change or are removed.
- `jndi-name` This attribute is used to assign a name for the cache in the JNDI name service.
- `module` This attribute configures the module whose class loader should be used when building this cache container's configuration.

The <abstract cache> abstract base element has the following child elements: **<indexing-properties>, <locking>**, **<transaction>**, **<eviction>**, **<expiration>**, **<store>**, **<file-store>**, **<string-keyed-jdbc-store>**, **<binary-keyed-jdbc-store>**, **<mixed-keyed-jdbc-store>**, **<remote-store>**.

<indexing-properties>
This child element defines properties to control indexing behaviour.
<locking>
This child element configures the locking behaviour of the cache.

- `isolation` This attribute the cache locking isolation level. Allowable values are NONE, SERIALIZABLE, REPEATABLE_READ, READ_COMMITTED, READ_UNCOMMITTED.
- `striping` If true, a pool of shared locks is maintained for all entries that need to be locked. Otherwise, a lock is created per entry in the cache. Lock striping helps control memory footprint but may reduce concurrency in the system.
- `acquire-timeout` This attribute configures the maximum time to attempt a particular lock acquisition.
- `concurrency-level` This attribute is used to configure the concurrency level. Adjust this value according to the number of concurrent threads interacting with Infinispan.

<transaction>
This child element configures the transactional behaviour of the cache.

- `mode` This attribute configures the transaction mode, setting the cache transaction mode to one of NONE, NON_XA, NON_DURABLE_XA, FULL_XA.
- `stop-timeout` If there are any ongoing transactions when a cache is stopped, Infinispan waits for ongoing remote and local transactions to finish. The amount of time to wait for is defined by the cache stop timeout.
- `locking` This attribute configures the locking mode for this cache, one of OPTIMISTIC or PESSIMISTIC.

<eviction>
This child element configures the eviction behaviour of the cache.

- `strategy` This attribute configures the cache eviction strategy. Available options are 'UNORDERED', 'FIFO', 'LRU', 'LIRS' and 'NONE' (to disable eviction).
- `max-entries` This attribute configures the maximum number of entries in a cache instance. If selected value is not a power of two the actual value will default to the least power of two larger than selected value. -1 means no limit.

<expiration>
This child element configures the expiration behaviour of the cache.

- `max-idle` This attribute configures the maximum idle time a cache entry will be maintained in the cache, in milliseconds. If the idle time is exceeded, the entry will be expired cluster-wide. -1 means the entries never expire.
- `lifespan` This attribute configures the maximum lifespan of a cache entry, after which the entry is expired cluster-wide, in milliseconds. -1 means the entries never expire.
- `interval` This attribute specifies the interval (in ms) between subsequent runs to purge expired entries from memory and any cache stores. If you wish to disable the periodic eviction process altogether, set wakeupInterval to -1.

The remaining child elements of the abstract base element **<cache>**, namely **<store>**, **<file-store>**, **<remote-store>**, **<string-keyed-jdbc-store>**, **<binary-keyed-jdbc-store>** and **<mixed-keyed-jdbc-store>**, each configures one of six key cache store types.

These cache store-related elements are actually part of an xsd extension hierarchy with abstract complexTypes **base-store** and **base-jdbc-store**. As before, in order to simplify the presentation, we notate these as pseudo-elements **<abstract base-store>** and **<abstract base-jdbc-store>**. In what follows, we first describe the extension hierarchy of base elements, and then show how the cache store elements relate to them.

<abstract base-store>
This abstract base element defines the attributes and child elements common to all cache stores.

- `shared` This attribute should be set to true when multiple cache instances share the same cache store (e.g. multiple nodes in a cluster using a JDBC-based CacheStore pointing to the same, shared database) Setting this to true avoids multiple cache instances writing the same modification multiple times. If enabled, only the node where the modification originated will write to the cache store. If disabled, each individual cache reacts to a potential remote update by storing the data to the cache store.
- `preload` This attribute configures whether or not, when the cache starts, data stored in the cache loader will be pre-loaded into memory. This is particularly useful when data in the cache loader is needed immediately after start-up and you want to avoid cache operations being delayed as a result of loading this data lazily. Can be used to provide a 'warm-cache' on start-up, however there is a performance penalty as start-up time is affected by this process. Note that pre-loading is done in a local fashion, so any data loaded is only stored locally in the node. No replication or distribution of the preloaded data happens. Also, Infinispan only pre-loads up to the maximum configured number of entries in eviction.
- `passivation` If true, data is only written to the cache store when it is evicted from memory, a phenomenon known as *passivation*. Next time the data is requested, it will be 'activated' which means that data will be brought back to memory and removed from the persistent store. If false, the cache store contains a copy of the cache contents in memory, so writes to cache result in cache store writes. This essentially gives you a 'write-through' configuration.
- `fetch-state` This attribute, if true, causes persistent state to be fetched when joining a cluster. If multiple cache stores are chained, only one of them can have this property enabled.
- `purge` This attribute configures whether the cache store is purged upon start-up.
- `singleton` This attribute configures whether or not the singleton store cache store is enabled. SingletonStore is a delegating cache store used for situations when only one instance in a cluster should interact with the underlying store.
- `class` This attribute configures a custom store implementation class to use for this cache store.
- `properties` This attribute is used to configure a list of cache store properties.

The abstract base element has one child element: **<write-behind>**
<write-behind>
This element is used to configure a cache store as write-behind instead of write-through. In write-through mode, writes to the cache are also *synchronously* written to the cache store, whereas in write-behind mode, writes to the cache are followed by *asynchronous* writes to the cache store.

- `flush-lock-timeout` This attribute configures the time-out for acquiring the lock which guards the state to be flushed to the cache store periodically.
- `modification-queue-size` This attribute configures the maximum number of entries in the asynchronous queue. When the queue is full, the store becomes write-through until it can accept new entries.
- `shutdown-timeout` This attribute configures the time-out (in ms) to stop the cache store.
- `thread-pool` This attribute is used to configure the size of the thread pool whose threads are responsible for applying the modifications to the cache store.

<abstract base-jdbc-store> extends <abstract base-store>
This abstract base element defines the attributes and child elements common to all JDBC-based cache stores.

- `datasource` This attribute configures the datasource for the JDBC-based cache store.
- `entry-table` This attribute configures the database table used to store cache entries.
- `bucket-table` This attribute configures the database table used to store binary cache entries.

<file-store> extends <abstract base-store>
This child element is used to configure a file-based cache store. This requires specifying the name of the file to be used as backing storage for the cache store.

- `relative-to` This attribute optionally configures a relative path prefix for the file store path. Can be null.
- `path` This attribute configures an absolute path to a file if **relative-to** is null; configures a relative path to the file, in relation to the value for **relative-to**, otherwise.

<remote-store> extends <abstract base-store>
This child element of cache is used to configure a remote cache store. It has a child <remote-servers>.

- `cache` This attribute configures the name of the remote cache to use for this remote store.
- `tcp-nodelay` This attribute configures a TCP_NODELAY value for communication with the remote cache.
- `socket-timeout` This attribute configures a socket time-out for communication with the remote cache.

<remote-servers>
This child element of cache configures a list of remote servers for this cache store.
<remote-server>
This element configures a remote server. A remote server is defined completely by a locally defined outbound socket binding, through which communication is made with the server.

- `outbound-socket-binding` This attribute configures an outbound socket binding for a remote server.

<local-cache> extends <abstract cache>

This element configures a local cache.

<abstract clustered-cache> extends <abstract cache>

This abstract base element defines the attributes and child elements common to all clustered caches. A clustered cache is a cache which spans multiple nodes in a cluster. It inherits from <cache>, so that all attributes and elements of <cache> are also defined for <clustered-cache>.

- `async-marshalling` This attribute configures async marshalling. If enabled, this will cause marshalling of entries to be performed asynchronously.
- `mode` This attribute configures the clustered cache mode, ASYNC for asynchronous operation, or SYNC for synchronous operation.
- `queue-size` In ASYNC mode, this attribute can be used to trigger flushing of the queue when it reaches a specific threshold.
- `queue-flush-interval` In ASYNC mode, this attribute controls how often the asynchronous thread used to flush the replication queue runs. This should be a positive integer which represents thread wakeup time in milliseconds.
- `remote-timeout` In SYNC mode, this attribute (in ms) used to wait for an acknowledgement when making a remote call, after which the call is aborted and an exception is thrown.

<invalidation-cache> extends <abstract clustered-cache>

This element configures an invalidation cache.

<abstract shared-cache> extends <abstract clustered-cache>

This abstract base element defines the attributes and child elements common to all shared caches. A shared cache is a clustered cache which shares state with its cache peers in the cluster. It inherits from <clustered-cache>, so that all attributes and elements of <clustered-cache> are also defined for <shared-cache>.

<state-transfer>

- `enabled` If enabled, this will cause the cache to ask neighbouring caches for state when it starts up, so the cache starts 'warm', although it will impact start-up time.
- `timeout` This attribute configures the maximum amount of time (ms) to wait for state from neighbouring caches, before throwing an exception and aborting start-up.
- `chunk-size` This attribute configures the size, in bytes, in which to batch the transfer of cache entries.

<backups>
<backup>

- `strategy` This attribute configures the backup strategy for this cache. Allowable values are SYNC, ASYNC.
- `failure-policy` This attribute configures the policy to follow when connectivity to the backup site fails. Allowable values are IGNORE, WARN, FAIL, CUSTOM.
- `enabled` This attribute configures whether or not this backup is enabled. If enabled, data will be sent to the backup site; otherwise, the backup site will be effectively ignored.
- `timeout` This attribute configures the time-out for replicating to the backup site.
- `after-failures` This attribute configures the number of failures after which this backup site should go off-line.
- `min-wait` This attribute configures the minimum time (in milliseconds) to wait after the max number of failures is reached, after which this backup site should go off-line.

<backup-for>

- `remote-cache` This attribute configures the name of the remote cache for which this cache acts as a backup.
- `remote-site` This attribute configures the site of the remote cache for which this cache acts as a backup.

<replicated-cache> extends <abstract shared-cache>

This element configures a replicated cache. With a replicated cache, all contents (key-value pairs) of the cache are replicated on all nodes in the cluster.

<distributed-cache> extends <abstract shared-cache>

This element configures a distributed cache. With a distributed cache, contents of the cache are selectively replicated on nodes in the cluster, according to the number of owners specified.

- `owners` This attribute configures the number of cluster-wide replicas for each cache entry.
- `segments` This attribute configures the number of hash space segments which is the granularity for key distribution in the cluster. Value must be strictly positive.
- `l1-lifespan` This attribute configures the maximum lifespan of an entry placed in the L1 cache. Configures the L1 cache behaviour in 'distributed' caches instances. In any other cache modes, this element is ignored.

Use Cases

In many cases, cache containers and caches will be configured via XML as in the example above, so that they will be available upon server start-up. However, cache containers and caches may also be added, removed or have their configurations changed in a running server by making use of the Wildfly management API command-line interface (CLI). In this section, we present some key use cases for the Infinispan management API.

The key use cases covered are:

- adding a cache container
- adding a cache to an existing cache container
- configuring the transaction subsystem of a cache

The Wildfly management API command-line interface (CLI) can be used to provide extensive information on the attributes and commands available in the Infinispan subsystem interface used in these examples.

Add a cache container

```
/subsystem=infinispan/cache-container=mycontainer:add(default-cache=<default-cache-name>)
/subsystem=infinispan/cache-container=mycontainer/transport=jgroups:add(lock-timeout=<timeout>)
```

Add a cache

```
/subsystem=infinispan/cache-container=mycontainer/local-cache=mylocalcache:add()
```

Configure the transaction component of a cache

```
/subsystem=infinispan/cache-container=mycontainer/local-cache=mylocalcache/component=transaction:a
```

7.15.10 mod_cluster Subsystem

The mod_cluster integration is done via the modcluster subsystem.

Configuration

Instance ID or JVMRoute

The instance-id or JVMRoute defaults to jboss.node.name property passed on server startup (e.g. via -Djboss.node.name=XYZ).

```
[standalone@localhost:9990 /] /subsystem=undertow/:read-attribute(name=instance-id)
{
    "outcome" => "success",
    "result" => expression "${jboss.node.name}"
}
```

To configure instance-id statically, configure the corresponding property in Undertow subsystem:

```
[standalone@localhost:9990 /]
/subsystem=undertow/:write-attribute(name=instance-id,value=myroute)
{
    "outcome" => "success",
    "response-headers" => {
        "operation-requires-reload" => true,
        "process-state" => "reload-required"
    }
}
```

Proxies

By default, mod_cluster is configured for multicast-based discovery. To specify a static list of proxies, create a remote-socket-binding for each proxy and then reference them in the 'proxies' attribute. See the following example for configuration in the domain mode:

```
[domain@localhost:9990 /]
/socket-binding-group=ha-sockets/remote-destination-outbound-socket-binding=proxy1:add(host=10.21.
port=6666)
{
    "outcome" => "success",
    "result" => undefined,
    "server-groups" => undefined
}
[domain@localhost:9990 /]
/socket-binding-group=ha-sockets/remote-destination-outbound-socket-binding=proxy2:add(host=10.21.
port=6666)
{
    "outcome" => "success",
    "result" => undefined,
    "server-groups" => undefined
}
[domain@localhost:9990 /]
/profile=ha/subsystem=modcluster/mod-cluster-config=configuration/:write-attribute(name=proxies,
value=[proxy1, proxy2]
{
    "outcome" => "success",
    "result" => undefined,
    "server-groups" => undefined
}
[domain@localhost:9990 /] :reload-servers
{
    "outcome" => "success",
    "result" => undefined,
    "server-groups" => undefined
}
```

Runtime Operations

The modcluster subsystem supports several operations:

```
[standalone@localhost:9999 subsystem=modcluster] :read-operation-names
{
    "outcome" => "success",
    "result" => [
        "add",
        "add-custom-metric",
        "add-metric",
        "add-proxy",
        "disable",
        "disable-context",
        "enable",
        "enable-context",
        "list-proxies",
        "read-attribute",
        "read-children-names",
        "read-children-resources",
        "read-children-types",
        "read-operation-description",
        "read-operation-names",
        "read-proxies-configuration",
        "read-proxies-info",
        "read-resource",
        "read-resource-description",
        "refresh",
        "remove-custom-metric",
        "remove-metric",
        "remove-proxy",
        "reset",
        "stop",
        "stop-context",
        "validate-address",
        "write-attribute"
    ]
}
```

The operations specific to the modcluster subsystem are divided in 3 categories the ones that affects the configuration and require a restart of the subsystem, the one that just modify the behaviour temporarily and the ones that display information from the httpd part.

operations displaying httpd informations

There are 2 operations that display how Apache httpd sees the node:

read-proxies-configuration

Send a DUMP message to all Apache httpd the node is connected to and display the message received from Apache httpd.

```
[standalone@localhost:9999 subsystem=modcluster] :read-proxies-configuration
{
    "outcome" => "success",
    "result" => [
        "neo3:6666",
        "balancer: [1] Name: mycluster Sticky: 1 [JSESSIONID]/[jsessionid] remove: 0 force: 1
Timeout: 0 Maxtry: 1
node: [1:1],Balancer: mycluster,JVMRoute: 498bb1f0-00d9-3436-a341-7f012bc2e7ec,Domain: [],Host:
127.0.0.1,Port: 8080,Type: http,flushpackets: 0,flushwait: 10,ping: 10,smax: 26,ttl: 60,timeout:
0
host: 1 [example.com] vhost: 1 node: 1
host: 2 [localhost] vhost: 1 node: 1
host: 3 [default-host] vhost: 1 node: 1
context: 1 [/myapp] vhost: 1 node: 1 status: 1
context: 2 [/] vhost: 1 node: 1 status: 1
",
        "jfcpc:6666",
        "balancer: [1] Name: mycluster Sticky: 1 [JSESSIONID]/[jsessionid] remove: 0 force: 1
Timeout: 0 maxAttempts: 1
node: [1:1],Balancer: mycluster,JVMRoute: 498bb1f0-00d9-3436-a341-7f012bc2e7ec,LBGroup: [],Host:
127.0.0.1,Port: 8080,Type: http,flushpackets: 0,flushwait: 10,ping: 10,smax: 26,ttl: 60,timeout:
0
host: 1 [default-host] vhost: 1 node: 1
host: 2 [localhost] vhost: 1 node: 1
host: 3 [example.com] vhost: 1 node: 1
context: 1 [/] vhost: 1 node: 1 status: 1
context: 2 [/myapp] vhost: 1 node: 1 status: 1
"
    ]
}
```

read-proxies-info

Send a INFO message to all Apache httpd the node is connected to and display the message received from Apache httpd.

```
[standalone@localhost:9999 subsystem=modcluster] :read-proxies-info
{
    "outcome" => "success",
    "result" => [
        "neo3:6666",
        "Node: [1],Name: 498bb1f0-00d9-3436-a341-7f012bc2e7ec,Balancer: mycluster,Domain: ,Host:
127.0.0.1,Port: 8080,Type: http,Flushpackets: Off,Flushwait: 10000,Ping: 10000000,Smax: 26,Ttl:
60000000,Elected: 0,Read: 0,Transfered: 0,Connected: 0,Load: -1
Vhost: [1:1:1], Alias: example.com
Vhost: [1:1:2], Alias: localhost
Vhost: [1:1:3], Alias: default-host
Context: [1:1:1], Context: /myapp, Status: ENABLED
Context: [1:1:2], Context: /, Status: ENABLED
",
        "jfcpc:6666",
        "Node: [1],Name: 498bb1f0-00d9-3436-a341-7f012bc2e7ec,Balancer: mycluster,LBGroup:
,Host: 127.0.0.1,Port: 8080,Type: http,Flushpackets: Off,Flushwait: 10,Ping: 10,Smax: 26,Ttl:
60,Elected: 0,Read: 0,Transfered: 0,Connected: 0,Load: 1
Vhost: [1:1:1], Alias: default-host
Vhost: [1:1:2], Alias: localhost
Vhost: [1:1:3], Alias: example.com
Context: [1:1:1], Context: /, Status: ENABLED
Context: [1:1:2], Context: /myapp, Status: ENABLED
"
    ]
}
```

operations that handle the proxies the node is connected too

there are 3 operation that could be used to manipulate the list of Apache httpd the node is connected too.

list-proxies:

Displays the httpd that are connected to the node. The httpd could be discovered via the Advertise protocol or via the proxy-list attribute.

```
[standalone@localhost:9999 subsystem=modcluster] :list-proxies
{
    "outcome" => "success",
    "result" => [
        "proxy1:6666",
        "proxy2:6666"
    ]
}
```

remove-proxy

Remove a proxy from the discovered proxies or temporarily from the proxy-list attribute.

```
[standalone@localhost:9999 subsystem=modcluster] :remove-proxy(host=jfcpc, port=6666)
{"outcome" => "success"}
```

proxy

Add a proxy to the discovered proxies or temporarily to the proxy-list attribute.

```
[standalone@localhost:9999 subsystem=modcluster] :add-proxy(host=jfcpc, port=6666)
{"outcome" => "success"}
```

Context related operations

Those operations allow to send context related commands to Apache httpd. They are send automatically when deploying or undeploying webapps.

enable-context

Tell Apache httpd that the context is ready receive requests.

```
[standalone@localhost:9999 subsystem=modcluster] :enable-context(context=/myapp,
virtualhost=default-host)
{"outcome" => "success"}
```

disable-context

Tell Apache httpd that it shouldn't send new session requests to the context of the virtualhost.

```
[standalone@localhost:9999 subsystem=modcluster] :disable-context(context=/myapp,
virtualhost=default-host)
{"outcome" => "success"}
```

stop-context

Tell Apache httpd that it shouldn't send requests to the context of the virtualhost.

```
[standalone@localhost:9999 subsystem=modcluster] :stop-context(context=/myapp,
virtualhost=default-host, waittime=50)
{"outcome" => "success"}
```

Node related operations

Those operations are like the context operation but they apply to all webapps running on the node and operation that affect the whole node.

refresh

Refresh the node by sending a new CONFIG message to Apache httpd.

reset

reset the connection between Apache httpd and the node

Configuration

Metric configuration

There are 4 metric operations corresponding to add and remove load metrics to the dynamic-load-provider. Note that when nothing is defined a simple-load-provider is use with a fixed load factor of one.

```
[standalone@localhost:9999 subsystem=modcluster] :read-resource(name=mod-cluster-config)
{
    "outcome" => "success",
    "result" => {"simple-load-provider" => {"factor" => "1"}}
}
```

that corresponds to the following configuration:

```
<subsystem xmlns="urn:jboss:domain:modcluster:1.0">
            <mod-cluster-config>
                <simple-load-provider factor="1"/>
            </mod-cluster-config>
 </subsystem>
```

metric

Add a metric to the dynamic-load-provider, the dynamic-load-provider in configuration is created if needed.

```
[standalone@localhost:9999 subsystem=modcluster] :add-metric(type=cpu)
{"outcome" => "success"}
[standalone@localhost:9999 subsystem=modcluster] :read-resource(name=mod-cluster-config)
{
    "outcome" => "success",
    "result" => {
        "dynamic-load-provider" => {
            "history" => 9,
            "decay" => 2,
            "load-metric" => [{
                "type" => "cpu"
            }]
        }
    }
}
```

remove-metric

Remove a metric from the dynamic-load-provider.

```
[standalone@localhost:9999 subsystem=modcluster] :remove-metric(type=cpu)
{"outcome" => "success"}
```

custom-metric / remove-custom-metric
like the add-metric and remove-metric except they require a class parameter instead the type. Usually they needed additional properties which can be specified

```
[standalone@localhost:9999 subsystem=modcluster] :add-custom-metric(class=myclass,
property=[("pro1" => "value1"), ("pro2" => "value2")]
{"outcome" => "success"}
```

which corresponds the following in the xml configuration file:

```
<subsystem xmlns="urn:jboss:domain:modcluster:1.0">
            <mod-cluster-config>
                <dynamic-load-provider history="9" decay="2">
                    <custom-load-metric class="myclass">
                        <property name="pro1" value="value1"/>
                        <property name="pro2" value="value2"/>
                    </custom-load-metric>
                </dynamic-load-provider>
            </mod-cluster-config>
</subsystem>
```

SSL Configuration using Elytron Subsystem

This document provides information how to configure mod_cluster subsystem to protect communication between mod_cluster and load balancer using SSL/TLS using Elytron Subsystem.

Overview

Elytron subsystem provides a powerful and flexible model to configure different security aspects for applications and the application server itself. At its core, Elytron subsystem exposes different capabilities to the application server in order centralize security related configuration in a single place and to allow other subsystems to consume these capabilities. One of the security capabilities exposed by Elytron subsystem is a Client `ssl-context` that can be used to configure mod_cluster subsystem to communicate with a load balancer using SSL/TLS.

When protecting the communication between the application server and the load balancer, you need do define a Client `ssl-context` in order to:

- Define a trust store holding the certificate chain that will be used to validate load balancer's certificate
- Define a trust manager to perform validations against the load balancer's certificate

Defining a Trust Store with the Trusted Certificates

To define a trust store in Elytron you can execute the following CLI command:

```
[standalone@localhost:9990 /] /subsystem=elytron/key-store=default-trust-store:add(type=JKS,
relative-to=jboss.server.config.dir, path=application.truststore,
credential-reference={clear-text=password})
```

In order to successfully execute the command above you must have a **application.truststore** file inside your **JBOSS_HOME/standalone/configuration** directory. Where the trust store is protected by a password with a value **password**. The trust store must contain the certificates associated with the load balancer or a certificate chain in case the load balancer's certificate is signed by a CA.

We strongly recommend you to avoid using self-signed certificates with your load balancer. Ideally, certificates should be signed by a CA and your trust store should contain a certificate chain representing your ROOT and Intermediary CAs.

Defining a Trust Manager To Validate Certificates

To define a trust manager in Elytron you can execute the following CLI command:

```
[standalone@localhost:9990 /]
/subsystem=elytron/trust-managers=default-trust-manager:add(algorithm=PKIX,
key-store=default-trust-store)
```

Here we are setting the **default-trust-store** as the source of the certificates that the application server trusts.

Defining a Client SSL Context and Configuring mod_cluster Subsystem

Finally, you can create the Client SSL Context that is going to be used by the mod_cluster subsystem when connecting to the load balancer using SSL/TLS:

```
[standalone@localhost:9990 /]
/subsystem=elytron/client-ssl-context=modcluster-client-ssl-context:add(trust-managers=default-tru
```

Now that the Client `ssl-context` is defined you can configure mod_cluster subsystem as follows:

```
[standalone@localhost:9990 /]
/subsystem=modcluster/mod-cluster-config=configuration:write-attribute(name=ssl-context,
value=modcluster-client-ssl-context)
```

Once you execute the last command above, reload the server:

```
[standalone@localhost:9990 /] :reload
```

Using a Certificate Revocation List

In case you want to validate the load balancer certificate against a Certificate Revocation List (CRL), you can configure the `trust-manager` in Elytron subsystem as follows:

```
[standalone@localhost:9990 /]
/subsystem=elytron/trust-managers=default-trust-manager:write-attribute(name=certificate-revocatio
value=intermediate.crl.pem)
```

To use a CRL your trust store must contain the certificate chain in order to check validity of both CRL list and the load balancer`s certificate.

A different way to configure a CRL is using the *Distribution Points* embedded in your certificates. For that, you need to configure a `certificate-revocation-list` as follows:

```
/subsystem=elytron/trust-managers=default-trust-manager:write-attribute(name=certificate-revocatio
```

7.16 HTTP Services

This section summarises the HTTP-based clustering features.

7.16.1 Subsystem Support

This section describes the key clustering subsystems, JGroups and Infinispan. Say a few words about how they work together.

JGroups Subsystem

Purpose

The JGroups subsystem provides group communication support for HA services in the form of JGroups channels.

Named channel instances permit application peers in a cluster to communicate as a group and in such a way that the communication satisfies defined properties (e.g. reliable, ordered, failure-sensitive). Communication properties are configurable for each channel and are defined by the protocol stack used to create the channel. Protocol stacks consist of a base transport layer (used to transport messages around the cluster) together with a user-defined, ordered stack of protocol layers, where each protocol layer supports a given communication property.

The JGroups subsystem provides the following features:

- allows definition of named protocol stacks
- view run-time metrics associated with channels
- specify a default stack for general use

In the following sections, we describe the JGroups subsystem.

JGroups channels are created transparently as part of the clustering functionality (e.g. on clustered application deployment, channels will be created behind the scenes to support clustered features such as session replication or transmission of SSO contexts around the cluster).

Configuration example

What follows is a sample JGroups subsystem configuration showing all of the possible elements and attributes which may be configured. We shall use this example to explain the meaning of the various elements and attributes.

The schema for the subsystem, describing all valid elements and attributes, can be found in the Wildfly distribution, in the docs/schema directory.

```
<subsystem xmlns="urn:jboss:domain:jgroups:5.0">
    <channels default="ee">
        <channel name="ee" stack="udp"/>
    </channels>
    <stacks>
        <stack name="udp">
            <transport type="UDP" socket-binding="jgroups-udp"/>
            <protocol type="PING"/>
            <protocol type="MERGE3"/>
            <protocol type="FD_SOCK"/>
            <protocol type="FD_ALL"/>
            <protocol type="VERIFY_SUSPECT"/>
            <protocol type="pbcast.NAKACK2"/>
            <protocol type="UNICAST3"/>
            <protocol type="pbcast.STABLE"/>
            <protocol type="pbcast.GMS"/>
            <protocol type="UFC"/>
            <protocol type="MFC"/>
            <protocol type="FRAG2"/>
        </stack>
        <stack name="tcp">
            <transport type="TCP" socket-binding="jgroups-tcp"/>
            <socket-protocol type="MPING" socket-binding="jgroups-mping"/>
            <protocol type="MERGE3"/>
            <protocol type="FD_SOCK"/>
            <protocol type="FD_ALL"/>
            <protocol type="VERIFY_SUSPECT"/>
            <protocol type="pbcast.NAKACK2"/>
            <protocol type="UNICAST3"/>
            <protocol type="pbcast.STABLE"/>
            <protocol type="pbcast.GMS"/>
            <protocol type="MFC"/>
            <protocol type="FRAG2"/>
        </stack>
    </stacks>
</subsystem>
```

<subsystem>

This element is used to configure the subsystem within a Wildfly system profile.

- `xmlns` This attribute specifies the XML namespace of the JGroups subsystem and, in particular, its version.
- `default-stack` This attribute is used to specify a default stack for the JGroups subsystem. This default stack will be used whenever a stack is required but no stack is specified.

<stack>

This element is used to configure a JGroups protocol stack.

- `name` This attribute is used to specify the name of the stack.

<transport>

This element is used to configure the transport layer (required) of the protocol stack.

- `type` This attribute specifies the transport type (e.g. UDP, TCP, TCPGOSSIP)
- `socket-binding` This attribute references a defined socket binding in the server profile. It is used when JGroups needs to create general sockets internally.
- `diagnostics-socket-binding` This attribute references a defined socket binding in the server profile. It is used when JGroups needs to create sockets for use with the diagnostics program. For more about the use of diagnostics, see the JGroups documentation for probe.sh.
- `default-executor` This attribute references a defined thread pool executor in the threads subsystem. It governs the allocation and execution of runnable tasks to handle incoming JGroups messages.
- `oob-executor` This attribute references a defined thread pool executor in the threads subsystem. It governs the allocation and execution of runnable tasks to handle incoming JGroups OOB (out-of-bound) messages.
- `timer-executor` This attribute references a defined thread pool executor in the threads subsystem. It governs the allocation and execution of runnable timer-related tasks.
- `shared` This attribute indicates whether or not this transport is shared amongst several JGroups stacks or not.
- `thread-factory` This attribute references a defined thread factory in the threads subsystem. It governs the allocation of threads for running tasks which are not handled by the executors above.
- `site` This attribute defines a site (data centre) id for this node.
- `rack` This attribute defines a rack (server rack) id for this node.
- `machine` This attribute defines a machine (host) is for this node.

site, rack and machine ids are used by the Infinispan topology-aware consistent hash function, which when using dist mode, prevents dist mode replicas from being stored on the same host, rack or site

.

<property>

This element is used to configure a transport property.

- `name` This attribute specifies the name of the protocol property. The value is provided as text for the property element.

<protocol>

This element is used to configure a (non-transport) protocol layer in the JGroups stack. Protocol layers are ordered within the stack.

- `type` This attribute specifies the name of the JGroups protocol implementation (e.g. MPING, pbcast.GMS), with the package prefix org.jgroups.protocols removed.
- `socket-binding` This attribute references a defined socket binding in the server profile. It is used when JGroups needs to create general sockets internally for this protocol instance.

<property>

This element is used to configure a protocol property.

- `name` This attribute specifies the name of the protocol property. The value is provided as text for the property element.

<relay>

This element is used to configure the RELAY protocol for a JGroups stack. RELAY is a protocol which provides cross-site replication between defined sites (data centres). In the RELAY protocol, defined sites specify the names of remote sites (backup sites) to which their data should be backed up. Channels are defined between sites to permit the RELAY protocol to transport the data from the current site to a backup site.

- `site` This attribute specifies the name of the current site. Site names can be referenced elsewhere (e.g. in the JGroups remote-site configuration elements, as well as backup configuration elements in the Infinispan subsystem)

<remote-site>

This element is used to configure a remote site for the RELAY protocol.

- `name` This attribute specifies the name of the remote site to which this configuration applies.
- `stack` This attribute specifies a JGroups protocol stack to use for communication between this site and the remote site.
- `cluster` This attribute specifies the name of the JGroups channel to use for communication between this site and the remote site.

Use Cases

In many cases, channels will be configured via XML as in the example above, so that the channels will be available upon server startup. However, channels may also be added, removed or have their configurations changed in a running server by making use of the Wildfly management API command-line interface (CLI). In this section, we present some key use cases for the JGroups management API.

The key use cases covered are:

- adding a stack
- adding a protocol to an existing stack
- adding a property to a protocol

The Wildfly management API command-line interface (CLI) itself can be used to provide extensive information on the attributes and commands available in the JGroups subsystem interface used in these examples.

Add a stack

```
/subsystem=jgroups/stack=mystack:add()
```

Add a protocol to a stack

```
/subsystem=jgroups/stack=mystack/transport=<type>:add(socket-binding=<socketbinding>)
```

```
/subsystem=jgroups/stack=mystack:protocol=<type>:add(socket-binding=<socketbinding>)
```

Add a property to a protocol

```
/subsystem=jgroups/stack=mystack/transport=<type>:map-put(name=properties, key=<property-name>,
value=<property-value>)
```

Infinispan Subsystem

Purpose

The Infinispan subsystem provides caching support for HA services in the form of Infinispan caches: high-performance, transactional caches which can operate in both non-distributed and distributed scenarios. Distributed caching support is used in the provision of many key HA services. For example, the failover of a session-oriented client HTTP request from a failing node to a new (failover) node depends on session data for the client being available on the new node. In other words, the client session data needs to be replicated across nodes in the cluster. This is effectively achieved via a distributed Infinispan cache. This approach to providing fail-over also applies to EJB SFSB sessions. Over and above providing support for fail-over, an underlying cache is also required when providing second-level caching for entity beans using Hibernate, and this case is also handled through the use of an Infinispan cache.

The Infinispan subsystem provides the following features:

- allows definition and configuration of named cache containers and caches
- view run-time metrics associated with cache container and cache instances

In the following sections, we describe the Infinispan subsystem.

Infiispan cache containers and caches are created transparently as part of the clustering functionality (e.g. on clustered application deployment, cache containers and their associated caches will be created behind the scenes to support clustered features such as session replication or caching of entities around the cluster).

Configuration Example

In this section, we provide an example XML configuration of the infinispan subsystem and review the configuration elements and attributes.

The schema for the subsystem, describing all valid elements and attributes, can be found in the Wildfly distribution, in the docs/schema directory.

```
<subsystem xmlns="urn:jboss:domain:infinispan:4.0">
  <cache-container name="server" aliases="singleton cluster" default-cache="default"
module="org.wildfly.clustering.server">
      <transport lock-timeout="60000"/>
      <replicated-cache name="default" mode="SYNC">
          <transaction mode="BATCH"/>      </replicated-cache>
  </cache-container>
  <cache-container name="web" default-cache="dist"
module="org.wildfly.clustering.web.infinispan">
      <transport lock-timeout="60000"/>
      <replicated-cache name="repl" mode="SYNC">
          <transaction mode="BATCH"/>          <file-store/>
      </replicated-cache>
      <distributed-cache name="dist" mode="SYNC">
          <transaction mode="BATCH"/>
          <file-store/>
      </distributed-cache>
  </cache-container>
  <cache-container name="ejb" aliases="sfsb" default-cache="dist"
module="org.wildfly.clustering.ejb.infinispan">
      <transport lock-timeout="60000"/>
      <replicated-cache name="repl" mode="SYNC">
          <transaction mode="BATCH"/>
          <file-store/>
      </replicated-cache>
      <distributed-cache name="dist" mode="SYNC" l1-lifespan="0">
          <transaction mode="BATCH"/>
          <file-store/>
      </distributed-cache>
  </cache-container>
  <cache-container name="hibernate" module="org.hibernate.infinispan">
      <transport lock-timeout="60000"/>
      <local-cache name="local-query">
          <transaction mode="NONE"/>
          <eviction strategy="LRU" max-entries="10000"/>
          <expiration max-idle="100000"/>
      </local-cache>
      <invalidation-cache name="entity" mode="SYNC">
          <transaction mode="NON_XA"/>
          <eviction strategy="LRU" max-entries="10000"/>
          <expiration max-idle="100000"/>
      </invalidation-cache>
      <replicated-cache name="timestamps" mode="ASYNC">
          <transaction mode="NONE"/>
          <eviction strategy="NONE"/>
       </replicated-cache>
  </cache-container>
</subsystem>
```

<cache-container>

This element is used to configure a cache container.

- `name` This attribute is used to specify the name of the cache container.
- `default-cache` This attribute configures the default cache to be used, when no cache is otherwise specified.
- `listener-executor` This attribute references a defined thread pool executor in the threads subsystem. It governs the allocation and execution of runnable tasks in the replication queue.
- `eviction-executor` This attribute references a defined thread pool executor in the threads subsystem. It governs the allocation and execution of runnable tasks to handle evictions.
- `replication-queue-executor` This attribute references a defined thread pool executor in the threads subsystem. It governs the allocation and execution of runnable tasks to handle asynchronous cache operations.
- `jndi-name` This attribute is used to assign a name for the cache container in the JNDI name service.
- `module` This attribute configures the module whose class loader should be used when building this cache container's configuration.
- `start` This attribute configured the cache container start mode and has since been deprecated, the only supported and the default value is LAZY (on-demand start).
- `aliases` This attribute is used to define aliases for the cache container name.

This element has the following child elements: **<transport>**, **<local-cache>**, **<invalidation-cache>**, **<replicated-cache>**, and **<distributed-cache>**.

<transport>

This element is used to configure the JGroups transport used by the cache container, when required.

- `channel` This attribute configures the JGroups channel to be used for the transport. If none is specified, the default channel as defined by the JGroups subsystem is used.
- `cluster` This attribute configures the name of the group communication cluster. This is the name which will be seen in debugging logs.
- `executor` This attribute references a defined thread pool executor in the threads subsystem. It governs the allocation and execution of runnable tasks to handle ? *<fill me in >*?.
- `lock-timeout` This attribute configures the time-out to be used when obtaining locks for the transport.
- `site` This attribute configures the site id of the cache container.
- `rack` This attribute configures the rack id of the cache container.
- `machine` This attribute configures the machine id of the cache container.

The presence of the transport element is required when operating in clustered mode

The remaining child elements of **<cache-container>**, namely **<local-cache>**, **<invalidation-cache>**, **<replicated-cache>** and **<distributed-cache>**, each configures one of four key cache types or classifications.

These cache-related elements are actually part of an xsd hierarchy with abstract complexTypes **cache**, **clustered-cache**, and **shared-cache**. In order to simplify the presentation, we notate these as pseudo-elements **<abstract cache>**, **<abstract clustered-cache>** and **<abstract shared-cache>**. In what follows, we first describe the extension hierarchy of base elements, and then show how the cache type elements relate to them.

<abstract cache>

This abstract base element defines the attributes and child elements common to all non-clustered caches.

- `name` This attribute configures the name of the cache. This name may be referenced by other subsystems.
- `start` This attribute configured the cache container start mode and has since been deprecated, the only supported and the default value is LAZY (on-demand start).
- `batching` This attribute configures batching. If enabled, the invocation batching API will be made available for this cache.
- `indexing` This attribute configures indexing. If enabled, entries will be indexed when they are added to the cache. Indexes will be updated as entries change or are removed.
- `jndi-name` This attribute is used to assign a name for the cache in the JNDI name service.
- `module` This attribute configures the module whose class loader should be used when building this cache container's configuration.

The <abstract cache> abstract base element has the following child elements: **<indexing-properties>, <locking>**, **<transaction>**, **<eviction>**, **<expiration>**, **<store>**, **<file-store>**, **<string-keyed-jdbc-store>**, **<binary-keyed-jdbc-store>**, **<mixed-keyed-jdbc-store>**, **<remote-store>**.

<indexing-properties>
This child element defines properties to control indexing behaviour.
<locking>
This child element configures the locking behaviour of the cache.

- `isolation` This attribute the cache locking isolation level. Allowable values are NONE, SERIALIZABLE, REPEATABLE_READ, READ_COMMITTED, READ_UNCOMMITTED.
- `striping` If true, a pool of shared locks is maintained for all entries that need to be locked. Otherwise, a lock is created per entry in the cache. Lock striping helps control memory footprint but may reduce concurrency in the system.
- `acquire-timeout` This attribute configures the maximum time to attempt a particular lock acquisition.
- `concurrency-level` This attribute is used to configure the concurrency level. Adjust this value according to the number of concurrent threads interacting with Infinispan.

<transaction>
This child element configures the transactional behaviour of the cache.

- `mode` This attribute configures the transaction mode, setting the cache transaction mode to one of NONE, NON_XA, NON_DURABLE_XA, FULL_XA.
- `stop-timeout` If there are any ongoing transactions when a cache is stopped, Infinispan waits for ongoing remote and local transactions to finish. The amount of time to wait for is defined by the cache stop timeout.
- `locking` This attribute configures the locking mode for this cache, one of OPTIMISTIC or PESSIMISTIC.

<eviction>
This child element configures the eviction behaviour of the cache.

- `strategy` This attribute configures the cache eviction strategy. Available options are 'UNORDERED', 'FIFO', 'LRU', 'LIRS' and 'NONE' (to disable eviction).
- `max-entries` This attribute configures the maximum number of entries in a cache instance. If selected value is not a power of two the actual value will default to the least power of two larger than selected value. -1 means no limit.

<expiration>
This child element configures the expiration behaviour of the cache.

- `max-idle` This attribute configures the maximum idle time a cache entry will be maintained in the cache, in milliseconds. If the idle time is exceeded, the entry will be expired cluster-wide. -1 means the entries never expire.
- `lifespan` This attribute configures the maximum lifespan of a cache entry, after which the entry is expired cluster-wide, in milliseconds. -1 means the entries never expire.
- `interval` This attribute specifies the interval (in ms) between subsequent runs to purge expired entries from memory and any cache stores. If you wish to disable the periodic eviction process altogether, set wakeupInterval to -1.

The remaining child elements of the abstract base element **<cache>**, namely **<store>**, **<file-store>**, **<remote-store>**, **<string-keyed-jdbc-store>**, **<binary-keyed-jdbc-store>** and **<mixed-keyed-jdbc-store>**, each configures one of six key cache store types.

These cache store-related elements are actually part of an xsd extension hierarchy with abstract complexTypes **base-store** and **base-jdbc-store**. As before, in order to simplify the presentation, we notate these as pseudo-elements **<abstract base-store>** and **<abstract base-jdbc-store>**. In what follows, we first describe the extension hierarchy of base elements, and then show how the cache store elements relate to them.

<abstract base-store>
This abstract base element defines the attributes and child elements common to all cache stores.

- `shared` This attribute should be set to true when multiple cache instances share the same cache store (e.g. multiple nodes in a cluster using a JDBC-based CacheStore pointing to the same, shared database) Setting this to true avoids multiple cache instances writing the same modification multiple times. If enabled, only the node where the modification originated will write to the cache store. If disabled, each individual cache reacts to a potential remote update by storing the data to the cache store.
- `preload` This attribute configures whether or not, when the cache starts, data stored in the cache loader will be pre-loaded into memory. This is particularly useful when data in the cache loader is needed immediately after start-up and you want to avoid cache operations being delayed as a result of loading this data lazily. Can be used to provide a 'warm-cache' on start-up, however there is a performance penalty as start-up time is affected by this process. Note that pre-loading is done in a local fashion, so any data loaded is only stored locally in the node. No replication or distribution of the preloaded data happens. Also, Infinispan only pre-loads up to the maximum configured number of entries in eviction.
- `passivation` If true, data is only written to the cache store when it is evicted from memory, a phenomenon known as *passivation*. Next time the data is requested, it will be 'activated' which means that data will be brought back to memory and removed from the persistent store. If false, the cache store contains a copy of the cache contents in memory, so writes to cache result in cache store writes. This essentially gives you a 'write-through' configuration.
- `fetch-state` This attribute, if true, causes persistent state to be fetched when joining a cluster. If multiple cache stores are chained, only one of them can have this property enabled.
- `purge` This attribute configures whether the cache store is purged upon start-up.
- `singleton` This attribute configures whether or not the singleton store cache store is enabled. SingletonStore is a delegating cache store used for situations when only one instance in a cluster should interact with the underlying store.
- `class` This attribute configures a custom store implementation class to use for this cache store.
- `properties` This attribute is used to configure a list of cache store properties.

The abstract base element has one child element: **<write-behind>**
<write-behind>
This element is used to configure a cache store as write-behind instead of write-through. In write-through mode, writes to the cache are also *synchronously* written to the cache store, whereas in write-behind mode, writes to the cache are followed by *asynchronous* writes to the cache store.

- `flush-lock-timeout` This attribute configures the time-out for acquiring the lock which guards the state to be flushed to the cache store periodically.
- `modification-queue-size` This attribute configures the maximum number of entries in the asynchronous queue. When the queue is full, the store becomes write-through until it can accept new entries.
- `shutdown-timeout` This attribute configures the time-out (in ms) to stop the cache store.
- `thread-pool` This attribute is used to configure the size of the thread pool whose threads are responsible for applying the modifications to the cache store.

<abstract base-jdbc-store> extends <abstract base-store>
This abstract base element defines the attributes and child elements common to all JDBC-based cache stores.

- `datasource` This attribute configures the datasource for the JDBC-based cache store.
- `entry-table` This attribute configures the database table used to store cache entries.
- `bucket-table` This attribute configures the database table used to store binary cache entries.

<file-store> extends <abstract base-store>
This child element is used to configure a file-based cache store. This requires specifying the name of the file to be used as backing storage for the cache store.

- `relative-to` This attribute optionally configures a relative path prefix for the file store path. Can be null.
- `path` This attribute configures an absolute path to a file if **relative-to** is null; configures a relative path to the file, in relation to the value for **relative-to**, otherwise.

<remote-store> extends <abstract base-store>
This child element of cache is used to configure a remote cache store. It has a child <remote-servers>.

- `cache` This attribute configures the name of the remote cache to use for this remote store.
- `tcp-nodelay` This attribute configures a TCP_NODELAY value for communication with the remote cache.
- `socket-timeout` This attribute configures a socket time-out for communication with the remote cache.

<remote-servers>
This child element of cache configures a list of remote servers for this cache store.
<remote-server>
This element configures a remote server. A remote server is defined completely by a locally defined outbound socket binding, through which communication is made with the server.

- `outbound-socket-binding` This attribute configures an outbound socket binding for a remote server.

<local-cache> extends <abstract cache>

This element configures a local cache.

<abstract clustered-cache> extends <abstract cache>

This abstract base element defines the attributes and child elements common to all clustered caches. A clustered cache is a cache which spans multiple nodes in a cluster. It inherits from <cache>, so that all attributes and elements of <cache> are also defined for <clustered-cache>.

- `async-marshalling` This attribute configures async marshalling. If enabled, this will cause marshalling of entries to be performed asynchronously.
- `mode` This attribute configures the clustered cache mode, ASYNC for asynchronous operation, or SYNC for synchronous operation.
- `queue-size` In ASYNC mode, this attribute can be used to trigger flushing of the queue when it reaches a specific threshold.
- `queue-flush-interval` In ASYNC mode, this attribute controls how often the asynchronous thread used to flush the replication queue runs. This should be a positive integer which represents thread wakeup time in milliseconds.
- `remote-timeout` In SYNC mode, this attribute (in ms) used to wait for an acknowledgement when making a remote call, after which the call is aborted and an exception is thrown.

<invalidation-cache> extends <abstract clustered-cache>

This element configures an invalidation cache.

<abstract shared-cache> extends <abstract clustered-cache>

This abstract base element defines the attributes and child elements common to all shared caches. A shared cache is a clustered cache which shares state with its cache peers in the cluster. It inherits from <clustered-cache>, so that all attributes and elements of <clustered-cache> are also defined for <shared-cache>.

<state-transfer>

- `enabled` If enabled, this will cause the cache to ask neighbouring caches for state when it starts up, so the cache starts 'warm', although it will impact start-up time.
- `timeout` This attribute configures the maximum amount of time (ms) to wait for state from neighbouring caches, before throwing an exception and aborting start-up.
- `chunk-size` This attribute configures the size, in bytes, in which to batch the transfer of cache entries.

<backups>
<backup>

- `strategy` This attribute configures the backup strategy for this cache. Allowable values are SYNC, ASYNC.
- `failure-policy` This attribute configures the policy to follow when connectivity to the backup site fails. Allowable values are IGNORE, WARN, FAIL, CUSTOM.
- `enabled` This attribute configures whether or not this backup is enabled. If enabled, data will be sent to the backup site; otherwise, the backup site will be effectively ignored.
- `timeout` This attribute configures the time-out for replicating to the backup site.
- `after-failures` This attribute configures the number of failures after which this backup site should go off-line.
- `min-wait` This attribute configures the minimum time (in milliseconds) to wait after the max number of failures is reached, after which this backup site should go off-line.

<backup-for>

- `remote-cache` This attribute configures the name of the remote cache for which this cache acts as a backup.
- `remote-site` This attribute configures the site of the remote cache for which this cache acts as a backup.

<replicated-cache> extends <abstract shared-cache>

This element configures a replicated cache. With a replicated cache, all contents (key-value pairs) of the cache are replicated on all nodes in the cluster.

<distributed-cache> extends <abstract shared-cache>

This element configures a distributed cache. With a distributed cache, contents of the cache are selectively replicated on nodes in the cluster, according to the number of owners specified.

- `owners` This attribute configures the number of cluster-wide replicas for each cache entry.
- `segments` This attribute configures the number of hash space segments which is the granularity for key distribution in the cluster. Value must be strictly positive.
- `l1-lifespan` This attribute configures the maximum lifespan of an entry placed in the L1 cache. Configures the L1 cache behaviour in 'distributed' caches instances. In any other cache modes, this element is ignored.

Use Cases

In many cases, cache containers and caches will be configured via XML as in the example above, so that they will be available upon server start-up. However, cache containers and caches may also be added, removed or have their configurations changed in a running server by making use of the Wildfly management API command-line interface (CLI). In this section, we present some key use cases for the Infinispan management API.

The key use cases covered are:

- adding a cache container
- adding a cache to an existing cache container
- configuring the transaction subsystem of a cache

The Wildfly management API command-line interface (CLI) can be used to provide extensive information on the attributes and commands available in the Infinispan subsystem interface used in these examples.

Add a cache container

```
/subsystem=infinispan/cache-container=mycontainer:add(default-cache=<default-cache-name>)
/subsystem=infinispan/cache-container=mycontainer/transport=jgroups:add(lock-timeout=<timeout>)
```

Add a cache

```
/subsystem=infinispan/cache-container=mycontainer/local-cache=mylocalcache:add()
```

Configure the transaction component of a cache

```
/subsystem=infinispan/cache-container=mycontainer/local-cache=mylocalcache/component=transaction:a
```

7.16.2 Clustered Web Sessions

7.16.3 Clustered SSO

7.16.4 Load Balancing

This section describes load balancing via Apache + mod_jk and Apache + mod_cluster.

7.16.5 Load balancing with Apache + mod_jk

Describe load balancing with Apache using mod_jk.

7.16.6 Load balancing with Apache + mod_cluster

Describe load balancing with Apache using mod_cluster.

mod_cluster Subsystem

The mod_cluster integration is done via the modcluster subsystem.

7.16.7 Configuration

Instance ID or JVMRoute

The instance-id or JVMRoute defaults to jboss.node.name property passed on server startup (e.g. via -Djboss.node.name=XYZ).

```
[standalone@localhost:9990 /] /subsystem=undertow/:read-attribute(name=instance-id)
{
    "outcome" => "success",
    "result" => expression "${jboss.node.name}"
}
```

To configure instance-id statically, configure the corresponding property in Undertow subsystem:

```
[standalone@localhost:9990 /]
/subsystem=undertow/:write-attribute(name=instance-id,value=myroute)
{
    "outcome" => "success",
    "response-headers" => {
        "operation-requires-reload" => true,
        "process-state" => "reload-required"
    }
}
```

Proxies

By default, mod_cluster is configured for multicast-based discovery. To specify a static list of proxies, create a remote-socket-binding for each proxy and then reference them in the 'proxies' attribute. See the following example for configuration in the domain mode:

```
[domain@localhost:9990 /]
/socket-binding-group=ha-sockets/remote-destination-outbound-socket-binding=proxy1:add(host=10.21.
port=6666)
{
    "outcome" => "success",
    "result" => undefined,
    "server-groups" => undefined
}
[domain@localhost:9990 /]
/socket-binding-group=ha-sockets/remote-destination-outbound-socket-binding=proxy2:add(host=10.21.
port=6666)
{
    "outcome" => "success",
    "result" => undefined,
    "server-groups" => undefined
}
[domain@localhost:9990 /]
/profile=ha/subsystem=modcluster/mod-cluster-config=configuration/:write-attribute(name=proxies,
value=[proxy1, proxy2]
{
    "outcome" => "success",
    "result" => undefined,
    "server-groups" => undefined
}
[domain@localhost:9990 /] :reload-servers
{
    "outcome" => "success",
    "result" => undefined,
    "server-groups" => undefined
}
```

7.16.8 Runtime Operations

The modcluster subsystem supports several operations:

```
[standalone@localhost:9999 subsystem=modcluster] :read-operation-names
{
    "outcome" => "success",
    "result" => [
        "add",
        "add-custom-metric",
        "add-metric",
        "add-proxy",
        "disable",
        "disable-context",
        "enable",
        "enable-context",
        "list-proxies",
        "read-attribute",
        "read-children-names",
        "read-children-resources",
        "read-children-types",
        "read-operation-description",
        "read-operation-names",
        "read-proxies-configuration",
        "read-proxies-info",
        "read-resource",
        "read-resource-description",
        "refresh",
        "remove-custom-metric",
        "remove-metric",
        "remove-proxy",
        "reset",
        "stop",
        "stop-context",
        "validate-address",
        "write-attribute"
    ]
}
```

The operations specific to the modcluster subsystem are divided in 3 categories the ones that affects the configuration and require a restart of the subsystem, the one that just modify the behaviour temporarily and the ones that display information from the httpd part.

operations displaying httpd informations

There are 2 operations that display how Apache httpd sees the node:

read-proxies-configuration

Send a DUMP message to all Apache httpd the node is connected to and display the message received from Apache httpd.

```
[standalone@localhost:9999 subsystem=modcluster] :read-proxies-configuration
{
    "outcome" => "success",
    "result" => [
        "neo3:6666",
        "balancer: [1] Name: mycluster Sticky: 1 [JSESSIONID]/[jsessionid] remove: 0 force: 1
Timeout: 0 Maxtry: 1
node: [1:1],Balancer: mycluster,JVMRoute: 498bb1f0-00d9-3436-a341-7f012bc2e7ec,Domain: [],Host:
127.0.0.1,Port: 8080,Type: http,flushpackets: 0,flushwait: 10,ping: 10,smax: 26,ttl: 60,timeout:
0
host: 1 [example.com] vhost: 1 node: 1
host: 2 [localhost] vhost: 1 node: 1
host: 3 [default-host] vhost: 1 node: 1
context: 1 [/myapp] vhost: 1 node: 1 status: 1
context: 2 [/] vhost: 1 node: 1 status: 1
",
        "jfcpc:6666",
        "balancer: [1] Name: mycluster Sticky: 1 [JSESSIONID]/[jsessionid] remove: 0 force: 1
Timeout: 0 maxAttempts: 1
node: [1:1],Balancer: mycluster,JVMRoute: 498bb1f0-00d9-3436-a341-7f012bc2e7ec,LBGroup: [],Host:
127.0.0.1,Port: 8080,Type: http,flushpackets: 0,flushwait: 10,ping: 10,smax: 26,ttl: 60,timeout:
0
host: 1 [default-host] vhost: 1 node: 1
host: 2 [localhost] vhost: 1 node: 1
host: 3 [example.com] vhost: 1 node: 1
context: 1 [/] vhost: 1 node: 1 status: 1
context: 2 [/myapp] vhost: 1 node: 1 status: 1
"
    ]
}
```

read-proxies-info

Send a INFO message to all Apache httpd the node is connected to and display the message received from Apache httpd.

```
[standalone@localhost:9999 subsystem=modcluster] :read-proxies-info
{
    "outcome" => "success",
    "result" => [
        "neo3:6666",
        "Node: [1],Name: 498bb1f0-00d9-3436-a341-7f012bc2e7ec,Balancer: mycluster,Domain: ,Host:
127.0.0.1,Port: 8080,Type: http,Flushpackets: Off,Flushwait: 10000,Ping: 10000000,Smax: 26,Ttl:
60000000,Elected: 0,Read: 0,Transfered: 0,Connected: 0,Load: -1
Vhost: [1:1:1], Alias: example.com
Vhost: [1:1:2], Alias: localhost
Vhost: [1:1:3], Alias: default-host
Context: [1:1:1], Context: /myapp, Status: ENABLED
Context: [1:1:2], Context: /, Status: ENABLED
",
        "jfcpc:6666",
        "Node: [1],Name: 498bb1f0-00d9-3436-a341-7f012bc2e7ec,Balancer: mycluster,LBGroup:
,Host: 127.0.0.1,Port: 8080,Type: http,Flushpackets: Off,Flushwait: 10,Ping: 10,Smax: 26,Ttl:
60,Elected: 0,Read: 0,Transfered: 0,Connected: 0,Load: 1
Vhost: [1:1:1], Alias: default-host
Vhost: [1:1:2], Alias: localhost
Vhost: [1:1:3], Alias: example.com
Context: [1:1:1], Context: /, Status: ENABLED
Context: [1:1:2], Context: /myapp, Status: ENABLED
"
    ]
}
```

operations that handle the proxies the node is connected too

there are 3 operation that could be used to manipulate the list of Apache httpd the node is connected too.

list-proxies:

Displays the httpd that are connected to the node. The httpd could be discovered via the Advertise protocol or via the proxy-list attribute.

```
[standalone@localhost:9999 subsystem=modcluster] :list-proxies
{
    "outcome" => "success",
    "result" => [
        "proxy1:6666",
        "proxy2:6666"
    ]
}
```

remove-proxy

Remove a proxy from the discovered proxies or temporarily from the proxy-list attribute.

```
[standalone@localhost:9999 subsystem=modcluster] :remove-proxy(host=jfcpc, port=6666)
{"outcome" => "success"}
```

proxy

Add a proxy to the discovered proxies or temporarily to the proxy-list attribute.

```
[standalone@localhost:9999 subsystem=modcluster] :add-proxy(host=jfcpc, port=6666)
{"outcome" => "success"}
```

Context related operations

Those operations allow to send context related commands to Apache httpd. They are send automatically when deploying or undeploying webapps.

enable-context

Tell Apache httpd that the context is ready receive requests.

```
[standalone@localhost:9999 subsystem=modcluster] :enable-context(context=/myapp,
virtualhost=default-host)
{"outcome" => "success"}
```

disable-context

Tell Apache httpd that it shouldn't send new session requests to the context of the virtualhost.

```
[standalone@localhost:9999 subsystem=modcluster] :disable-context(context=/myapp,
virtualhost=default-host)
{"outcome" => "success"}
```

stop-context

Tell Apache httpd that it shouldn't send requests to the context of the virtualhost.

```
[standalone@localhost:9999 subsystem=modcluster] :stop-context(context=/myapp,
virtualhost=default-host, waittime=50)
{"outcome" => "success"}
```

Node related operations

Those operations are like the context operation but they apply to all webapps running on the node and operation that affect the whole node.

refresh

Refresh the node by sending a new CONFIG message to Apache httpd.

reset

reset the connection between Apache httpd and the node

Configuration

Metric configuration

There are 4 metric operations corresponding to add and remove load metrics to the dynamic-load-provider. Note that when nothing is defined a simple-load-provider is use with a fixed load factor of one.

```
[standalone@localhost:9999 subsystem=modcluster] :read-resource(name=mod-cluster-config)
{
    "outcome" => "success",
    "result" => {"simple-load-provider" => {"factor" => "1"}}
}
```

that corresponds to the following configuration:

```
<subsystem xmlns="urn:jboss:domain:modcluster:1.0">
            <mod-cluster-config>
                <simple-load-provider factor="1"/>
            </mod-cluster-config>
 </subsystem>
```

metric

Add a metric to the dynamic-load-provider, the dynamic-load-provider in configuration is created if needed.

```
[standalone@localhost:9999 subsystem=modcluster] :add-metric(type=cpu)
{"outcome" => "success"}
[standalone@localhost:9999 subsystem=modcluster] :read-resource(name=mod-cluster-config)
{
    "outcome" => "success",
    "result" => {
        "dynamic-load-provider" => {
            "history" => 9,
            "decay" => 2,
            "load-metric" => [{
                "type" => "cpu"
            }]
        }
    }
}
```

remove-metric

Remove a metric from the dynamic-load-provider.

```
[standalone@localhost:9999 subsystem=modcluster] :remove-metric(type=cpu)
{"outcome" => "success"}
```

custom-metric / remove-custom-metric

like the add-metric and remove-metric except they require a class parameter instead the type. Usually they needed additional properties which can be specified

```
[standalone@localhost:9999 subsystem=modcluster] :add-custom-metric(class=myclass,
property=[("pro1" => "value1"), ("pro2" => "value2")]
{"outcome" => "success"}
```

which corresponds the following in the xml configuration file:

```
<subsystem xmlns="urn:jboss:domain:modcluster:1.0">
            <mod-cluster-config>
                <dynamic-load-provider history="9" decay="2">
                    <custom-load-metric class="myclass">
                        <property name="pro1" value="value1"/>
                        <property name="pro2" value="value2"/>
                    </custom-load-metric>
                </dynamic-load-provider>
            </mod-cluster-config>
</subsystem>
```

7.16.9 Clustered Web Sessions

7.16.10 Clustered SSO

7.16.11 Load Balancing

This section describes load balancing via Apache + mod_jk and Apache + mod_cluster.

Load balancing with Apache + mod_jk

Describe load balancing with Apache using mod_jk.

Load balancing with Apache + mod_cluster

Describe load balancing with Apache using mod_cluster.

mod_cluster Subsystem

The mod_cluster integration is done via the modcluster subsystem.

Configuration

Instance ID or JVMRoute

The instance-id or JVMRoute defaults to jboss.node.name property passed on server startup (e.g. via -Djboss.node.name=XYZ).

```
[standalone@localhost:9990 /] /subsystem=undertow/:read-attribute(name=instance-id)
{
    "outcome" => "success",
    "result" => expression "${jboss.node.name}"
}
```

To configure instance-id statically, configure the corresponding property in Undertow subsystem:

```
[standalone@localhost:9990 /]
/subsystem=undertow/:write-attribute(name=instance-id,value=myroute)
{
    "outcome" => "success",
    "response-headers" => {
        "operation-requires-reload" => true,
        "process-state" => "reload-required"
    }
}
```

Proxies

By default, mod_cluster is configured for multicast-based discovery. To specify a static list of proxies, create a remote-socket-binding for each proxy and then reference them in the 'proxies' attribute. See the following example for configuration in the domain mode:

```
[domain@localhost:9990 /]
/socket-binding-group=ha-sockets/remote-destination-outbound-socket-binding=proxy1:add(host=10.21.
port=6666)
{
    "outcome" => "success",
    "result" => undefined,
    "server-groups" => undefined
}
[domain@localhost:9990 /]
/socket-binding-group=ha-sockets/remote-destination-outbound-socket-binding=proxy2:add(host=10.21.
port=6666)
{
    "outcome" => "success",
    "result" => undefined,
    "server-groups" => undefined
}
[domain@localhost:9990 /]
/profile=ha/subsystem=modcluster/mod-cluster-config=configuration/:write-attribute(name=proxies,
value=[proxy1, proxy2]
{
    "outcome" => "success",
    "result" => undefined,
    "server-groups" => undefined
}
[domain@localhost:9990 /] :reload-servers
{
    "outcome" => "success",
    "result" => undefined,
    "server-groups" => undefined
}
```

Runtime Operations

The modcluster subsystem supports several operations:

```
[standalone@localhost:9999 subsystem=modcluster] :read-operation-names
{
    "outcome" => "success",
    "result" => [
        "add",
        "add-custom-metric",
        "add-metric",
        "add-proxy",
        "disable",
        "disable-context",
        "enable",
        "enable-context",
        "list-proxies",
        "read-attribute",
        "read-children-names",
        "read-children-resources",
        "read-children-types",
        "read-operation-description",
        "read-operation-names",
        "read-proxies-configuration",
        "read-proxies-info",
        "read-resource",
        "read-resource-description",
        "refresh",
        "remove-custom-metric",
        "remove-metric",
        "remove-proxy",
        "reset",
        "stop",
        "stop-context",
        "validate-address",
        "write-attribute"
    ]
}
```

The operations specific to the modcluster subsystem are divided in 3 categories the ones that affects the configuration and require a restart of the subsystem, the one that just modify the behaviour temporarily and the ones that display information from the httpd part.

operations displaying httpd informations

There are 2 operations that display how Apache httpd sees the node:

read-proxies-configuration

Send a DUMP message to all Apache httpd the node is connected to and display the message received from Apache httpd.

```
[standalone@localhost:9999 subsystem=modcluster] :read-proxies-configuration
{
    "outcome" => "success",
    "result" => [
        "neo3:6666",
        "balancer: [1] Name: mycluster Sticky: 1 [JSESSIONID]/[jsessionid] remove: 0 force: 1
Timeout: 0 Maxtry: 1
node: [1:1],Balancer: mycluster,JVMRoute: 498bb1f0-00d9-3436-a341-7f012bc2e7ec,Domain: [],Host:
127.0.0.1,Port: 8080,Type: http,flushpackets: 0,flushwait: 10,ping: 10,smax: 26,ttl: 60,timeout:
0
host: 1 [example.com] vhost: 1 node: 1
host: 2 [localhost] vhost: 1 node: 1
host: 3 [default-host] vhost: 1 node: 1
context: 1 [/myapp] vhost: 1 node: 1 status: 1
context: 2 [/] vhost: 1 node: 1 status: 1
",
        "jfcpc:6666",
        "balancer: [1] Name: mycluster Sticky: 1 [JSESSIONID]/[jsessionid] remove: 0 force: 1
Timeout: 0 maxAttempts: 1
node: [1:1],Balancer: mycluster,JVMRoute: 498bb1f0-00d9-3436-a341-7f012bc2e7ec,LBGroup: [],Host:
127.0.0.1,Port: 8080,Type: http,flushpackets: 0,flushwait: 10,ping: 10,smax: 26,ttl: 60,timeout:
0
host: 1 [default-host] vhost: 1 node: 1
host: 2 [localhost] vhost: 1 node: 1
host: 3 [example.com] vhost: 1 node: 1
context: 1 [/] vhost: 1 node: 1 status: 1
context: 2 [/myapp] vhost: 1 node: 1 status: 1
"
    ]
}
```

read-proxies-info

Send a INFO message to all Apache httpd the node is connected to and display the message received from Apache httpd.

```
[standalone@localhost:9999 subsystem=modcluster] :read-proxies-info
{
    "outcome" => "success",
    "result" => [
        "neo3:6666",
        "Node: [1],Name: 498bb1f0-00d9-3436-a341-7f012bc2e7ec,Balancer: mycluster,Domain: ,Host:
127.0.0.1,Port: 8080,Type: http,Flushpackets: Off,Flushwait: 10000,Ping: 10000000,Smax: 26,Ttl:
60000000,Elected: 0,Read: 0,Transfered: 0,Connected: 0,Load: -1
Vhost: [1:1:1], Alias: example.com
Vhost: [1:1:2], Alias: localhost
Vhost: [1:1:3], Alias: default-host
Context: [1:1:1], Context: /myapp, Status: ENABLED
Context: [1:1:2], Context: /, Status: ENABLED
",
        "jfcpc:6666",
        "Node: [1],Name: 498bb1f0-00d9-3436-a341-7f012bc2e7ec,Balancer: mycluster,LBGroup:
,Host: 127.0.0.1,Port: 8080,Type: http,Flushpackets: Off,Flushwait: 10,Ping: 10,Smax: 26,Ttl:
60,Elected: 0,Read: 0,Transfered: 0,Connected: 0,Load: 1
Vhost: [1:1:1], Alias: default-host
Vhost: [1:1:2], Alias: localhost
Vhost: [1:1:3], Alias: example.com
Context: [1:1:1], Context: /, Status: ENABLED
Context: [1:1:2], Context: /myapp, Status: ENABLED
"
    ]
}
```

operations that handle the proxies the node is connected too

there are 3 operation that could be used to manipulate the list of Apache httpd the node is connected too.

list-proxies:

Displays the httpd that are connected to the node. The httpd could be discovered via the Advertise protocol or via the proxy-list attribute.

```
[standalone@localhost:9999 subsystem=modcluster] :list-proxies
{
    "outcome" => "success",
    "result" => [
        "proxy1:6666",
        "proxy2:6666"
    ]
}
```

remove-proxy

Remove a proxy from the discovered proxies or temporarily from the proxy-list attribute.

```
[standalone@localhost:9999 subsystem=modcluster] :remove-proxy(host=jfcpc, port=6666)
{"outcome" => "success"}
```

proxy

Add a proxy to the discovered proxies or temporarily to the proxy-list attribute.

```
[standalone@localhost:9999 subsystem=modcluster] :add-proxy(host=jfcpc, port=6666)
{"outcome" => "success"}
```

Context related operations

Those operations allow to send context related commands to Apache httpd. They are send automatically when deploying or undeploying webapps.

enable-context

Tell Apache httpd that the context is ready receive requests.

```
[standalone@localhost:9999 subsystem=modcluster] :enable-context(context=/myapp,
virtualhost=default-host)
{"outcome" => "success"}
```

disable-context

Tell Apache httpd that it shouldn't send new session requests to the context of the virtualhost.

```
[standalone@localhost:9999 subsystem=modcluster] :disable-context(context=/myapp,
virtualhost=default-host)
{"outcome" => "success"}
```

stop-context

Tell Apache httpd that it shouldn't send requests to the context of the virtualhost.

```
[standalone@localhost:9999 subsystem=modcluster] :stop-context(context=/myapp,
virtualhost=default-host, waittime=50)
{"outcome" => "success"}
```

Node related operations

Those operations are like the context operation but they apply to all webapps running on the node and operation that affect the whole node.

refresh

Refresh the node by sending a new CONFIG message to Apache httpd.

reset

reset the connection between Apache httpd and the node

Configuration

Metric configuration

There are 4 metric operations corresponding to add and remove load metrics to the dynamic-load-provider. Note that when nothing is defined a simple-load-provider is use with a fixed load factor of one.

```
[standalone@localhost:9999 subsystem=modcluster] :read-resource(name=mod-cluster-config)
{
    "outcome" => "success",
    "result" => {"simple-load-provider" => {"factor" => "1"}}
}
```

that corresponds to the following configuration:

```
<subsystem xmlns="urn:jboss:domain:modcluster:1.0">
            <mod-cluster-config>
                <simple-load-provider factor="1"/>
            </mod-cluster-config>
 </subsystem>
```

metric

Add a metric to the dynamic-load-provider, the dynamic-load-provider in configuration is created if needed.

```
[standalone@localhost:9999 subsystem=modcluster] :add-metric(type=cpu)
{"outcome" => "success"}
[standalone@localhost:9999 subsystem=modcluster] :read-resource(name=mod-cluster-config)
{
    "outcome" => "success",
    "result" => {
        "dynamic-load-provider" => {
            "history" => 9,
            "decay" => 2,
            "load-metric" => [{
                "type" => "cpu"
            }]
        }
    }
}
```

remove-metric

Remove a metric from the dynamic-load-provider.

```
[standalone@localhost:9999 subsystem=modcluster] :remove-metric(type=cpu)
{"outcome" => "success"}
```

custom-metric / remove-custom-metric
like the add-metric and remove-metric except they require a class parameter instead the type. Usually they needed additional properties which can be specified

```
[standalone@localhost:9999 subsystem=modcluster] :add-custom-metric(class=myclass,
property=[("pro1" => "value1"), ("pro2" => "value2")]
{"outcome" => "success"}
```

which corresponds the following in the xml configuration file:

```
<subsystem xmlns="urn:jboss:domain:modcluster:1.0">
            <mod-cluster-config>
                <dynamic-load-provider history="9" decay="2">
                    <custom-load-metric class="myclass">
                        <property name="pro1" value="value1"/>
                        <property name="pro2" value="value2"/>
                    </custom-load-metric>
                </dynamic-load-provider>
            </mod-cluster-config>
</subsystem>
```

Apache httpd

The recommended front-end module is mod_cluster but mod_jk or mod_proxy could be used as in Tomcat or other AS version.

To use AJP define a ajp connector in the web subsystem like:

<subsystem xmlns="urn:jboss:domain:web:1.0">
        <connector name="http" protocol="HTTP/1.1" socket-binding="http"/>
        <connector name="ajp" protocol="AJP/1.3" socket-binding="ajp"/>

To the ajp in the in the socket-binding-group like:

<socket-binding-group name="standard-sockets" default-interface="public">
      ....
        <socket-binding name="http" port="8080"/>
        <socket-binding name="ajp" port="8009"/>
        <socket-binding name="https" port="8443"/>

7.17 EJB Services

This chapter explains how clustering of EJBs works in WildFly 8.

7.17.1 EJB Subsystem

7.17.2 EJB Timer

Wildfly now supports clustered database backed timers. For details have a look to the EJB3 reference section

Marking an EJB as clustered

WildFly 8 allows clustering of stateful session beans. A stateful session bean can be marked with `@org.jboss.ejb3.annotation.Clustered` annotation or be marked as clustered using the jboss-ejb3.xml's `<clustered>` element.

MyStatefulBean

```
import org.jboss.ejb3.annotation.Clustered;
import javax.ejb.Stateful;

@Stateful
@Clustered
public class MyStatefulBean {
...
}
```

jboss-ejb3.xml

```
<jboss xmlns="http://www.jboss.com/xml/ns/javaee"
       xmlns:jee="http://java.sun.com/xml/ns/javaee"
       xmlns:c="urn:clustering:1.0">


    <jee:assembly-descriptor>
        <c:clustering>
            <jee:ejb-name>DDBasedClusteredBean</jee:ejb-name>
            <c:clustered>true</c:clustered>
        </c:clustering>
    </jee:assembly-descriptor>
</jboss>
```

Deploying clustered EJBs

Clustering support is available in the HA profiles of WildFly 8. In this chapter we'll be using the standalone server for explaining the details. However, the same applies to servers in a domain mode. Starting the standalone server with HA capabilities enabled, involves starting it with the standalone-ha.xml (or even standalone-full-ha.xml):

```
./standalone.sh -server-config=standalone-ha.xml
```

This will start a single instance of the server with HA capabilities. Deploying the EJBs to this instance *doesn't* involve anything special and is the same as explained in the application deployment chapter.

Obviously, to be able to see the benefits of clustering, you'll need more than one instance of the server. So let's start another server with HA capabilities. That another instance of the server can either be on the same machine or on some other machine. If it's on the same machine, the two things you have to make sure is that you pass the port offset for the second instance and also make sure that each of the server instances have a unique `jboss.node.name` system property. You can do that by passing the following two system properties to the startup command:

```
./standalone.sh -server-config=standalone-ha.xml -Djboss.socket.binding.port-offset=<offset of
your choice> -Djboss.node.name=<unique node name>
```

Follow whichever approach you feel comfortable with for deploying the EJB deployment to this instance too.

Deploying the application on just one node of a standalone instance of a clustered server does **not** mean that it will be automatically deployed to the other clustered instance. You will have to do deploy it explicitly on the other standalone clustered instance too. Or you can start the servers in domain mode so that the deployment can be deployed to all the server within a server group. See the admin guide for more details on domain setup.

Now that you have deployed an application with clustered EJBs on both the instances, the EJBs are now capable of making use of the clustering features.

Failover for clustered EJBs

Clustered EJBs have failover capability. The state of the @Stateful @Clustered EJBs is replicated across the cluster nodes so that if one of the nodes in the cluster goes down, some other node will be able to take over the invocations. Let's see how it's implemented in WildFly 8. In the next few sections we'll see how it works for remote (standalone) clients and for clients in another remote WildFly server instance. Although, there isn't a difference in how it works in both these cases, we'll still explain it separately so as to make sure there aren't any unanswered questions.

Remote standalone clients

In this section we'll consider a remote standalone client (i.e. a client which runs in a separate JVM and *isn't* running within another WildFly 8 instance). Let's consider that we have 2 servers, server X and server Y which we started earlier. Each of these servers has the clustered EJB deployment. A standalone remote client can use either the JNDI approach or native JBoss EJB client APIs to communicate with the servers. The important thing to note is that when you are invoking clustered EJB deployments, you do **not** have to list all the servers within the cluster (which obviously wouldn't have been feasible due the dynamic nature of cluster node additions within a cluster).

The remote client just has to list only one of the servers with the clustering capability. In this case, we can either list server X (in jboss-ejb-client.properties) *or* server Y. This server will act as the starting point for cluster topology communication between the client and the clustered nodes.

Note that you have to configure the *ejb* cluster in the jboss-ejb-client.properties configuration file, like so:

```
remote.clusters=ejb
remote.cluster.ejb.connect.options.org.xnio.Options.SASL_POLICY_NOANONYMOUS=false
remote.cluster.ejb.connect.options.org.xnio.Options.SSL_ENABLED=false
```

Cluster topology communication

When a client connects to a server, the JBoss EJB client implementation (internally) communicates with the server for cluster topology information, if the server had clustering capability. In our example above, let's assume we listed server X as the initial server to connect to. When the client connects to server X, the server will send back an (asynchronous) cluster topology message to the client. This topology message consists of the cluster name(s) and the information of the nodes that belong to the cluster. The node information includes the node address and port number to connect to (whenever necessary). So in this example, the server X will send back the cluster topology consisting of the other server Y which belongs to the cluster.

In case of stateful (clustered) EJBs, a typical invocation flow involves creating of a session for the stateful bean, which happens when you do a JNDI lookup for that bean, and then invoking on the returned proxy. The lookup for stateful bean, internally, triggers a (synchronous) session creation request from the client to the server. In this case, the session creation request goes to server X since that's the initial connection that we have configured in our jboss-ejb-client.properties. Since server X is clustered, it will return back a session id and along with send back an *"affinity"* of that session. In case of clustered servers, the affinity equals to the name of the cluster to which the stateful bean belongs on the server side. For non-clustered beans, the affinity is just the node name on which the session was created. This *affinity* will later help the EJB client to route the invocations on the proxy, appropriately to either a node within a cluster (for clustered beans) or to a specific node (for non-clustered beans). While this session creation request is going on, the server X will also send back an asynchronous message which contains the cluster topology. The JBoss EJB client implementation will take note of this topology information and will later use it for connection creation to nodes within the cluster and routing invocations to those nodes, whenever necessary.

Now that we know how the cluster topology information is communicated from the server to the client, let see how failover works. Let's continue with the example of server X being our starting point and a client application looking up a stateful bean and invoking on it. During these invocations, the client side will have collected the cluster topology information from the server. Now let's assume for some reason, server X goes down and the client application subsequent invokes on the proxy. The JBoss EJB client implementation, at this stage will be aware of the affinity and in this case it's a cluster affinity. Because of the cluster topology information it has, it knows that the cluster has two nodes server X and server Y. When the invocation now arrives, it sees that the server X is down. So it uses a selector to fetch a suitable node from among the cluster nodes. The selector itself is configurable, but we'll leave it from discussion for now. When the selector returns a node from among the cluster, the JBoss EJB client implementation creates a connection to that node (if not already created earlier) and creates a EJB receiver out of it. Since in our example, the only other node in the cluster is server Y, the selector will return that node and the JBoss EJB client implementation will use it to create a EJB receiver out of it and use that receiver to pass on the invocation on the proxy. Effectively, the invocation has now failed over to a different node within the cluster.

Remote clients on another instance of WildFly 8

So far we discussed remote standalone clients which typically use either the EJB client API or the jboss-ejb-client.properties based approach to configure and communicate with the servers where the clustered beans are deployed. Now let's consider the case where the client is an application deployed another AS7 instance and it wants to invoke on a clustered stateful bean which is deployed on another instance of WildFly 8. In this example let's consider a case where we have 3 servers involved. Server X and Server Y both belong to a cluster and have clustered EJB deployed on them. Let's consider another server instance Server C (which may or may *not* have clustering capability) which acts as a client on which there's a deployment which wants to invoke on the clustered beans deployed on server X and Y and achieve failover.

The configurations required to achieve this are explained in this chapter. As you can see the configurations are done in a jboss-ejb-client.xml which points to a remote outbound connection to the other server. This jboss-ejb-client.xml goes in the deployment of server C (since that's our client). As explained eariler, the client configuration need **not** point to all clustered nodes. Instead it just has to point to one of them which will act as a start point for communication. So in this case, we can create a remote outbound connection on server C to server X and use server X as our starting point for communication. Just like in the case of remote standalone clients, when the application on server C (client) looks up a stateful bean, a session creation request will be sent to server X which will send back a session id and the cluster affinity for it. Furthermore, server X asynchronously send back a message to server C (client) containing the cluster topology. This topology information will include the node information of server Y (since that belongs to the cluster along with server X). Subsequent invocations on the proxy will be routed appropriately to the nodes in the cluster. If server X goes down, as explained earlier, a different node from the cluster will be selected and the invocation will be forwarded to that node.

As can be seen both remote standalone client and remote clients on another WildFly 8 instance act similar in terms of failover.

Testcases for failover of stateful beans

We have testcases in WildFly 8 testsuite which test that whatever is explained above works as expected. The RemoteEJBClientStatefulBeanFailoverTestCase tests the case where a stateful EJB uses @Clustered annotation to mark itself as clustered. We also have RemoteEJBClientDDBasedSFSBFailoverTestCase which uses jboss-ejb3.xml to mark a stateful EJB as clustered. Both these testcases test that when a node goes down in a cluster, the client invocation is routed to a different node in the cluster.

7.17.3 EJB Timer

Wildfly now supports clustered database backed timers. For details have a look to the EJB3 reference section

7.18 HA Singleton Features

In general, an HA or clustered singleton is a service that exists on multiple nodes in a cluster, but is active on just a single node at any given time. If the node providing the service fails or is shut down, a new singleton provider is chosen and started. Thus, other than a brief interval when one provider has stopped and another has yet to start, the service is always running on one node.

7.18.1 Singleton subsystem

WildFly 10 introduces a “singleton” subsystem, which defines a set of policies that define how an HA singleton should behave. A singleton policy can be used to instrument singleton deployments or to create singleton MSC services.

Configuration

The default subsystem configuration from WildFly’s ha and full-ha profile looks like:

```
<subsystem xmlns="urn:jboss:domain:singleton:1.0">
    <singleton-policies default="default">
        <singleton-policy name="default" cache-container="server">
            <simple-election-policy/>
        </singleton-policy>
    </singleton-policies>
</subsystem>
```

A singleton policy defines:

1. A unique name
2. A cache container and cache with which to register singleton provider candidates
3. An election policy
4. A quorum (optional)

One can add a new singleton policy via the following management operation:

```
/subsystem=singleton/singleton-policy=foo:add(cache-container=server)
```

Cache configuration

The cache-container and cache attributes of a singleton policy must reference a valid cache from the Infinispan subsystem. If no specific cache is defined, the default cache of the cache container is assumed. This cache is used as a registry of which nodes can provide a given service and will typically use a replicated-cache configuration.

Election policies

WildFly 10 includes 2 singleton election policy implementations:

- **simple**
  Elects the provider (a.k.a. master) of a singleton service based on a specified position in a circular linked list of eligible nodes sorted by descending age. Position=0, the default value, refers to the oldest node, 1 is second oldest, etc. ; while position=-1 refers to the youngest node, -2 to the second youngest, etc.
  e.g.

```
/subsystem=singleton/singleton-policy=foo/election-policy=simple:add(position=-1)
```

- **random**
  Elects a random member to be the provider of a singleton service
  e.g.

```
/subsystem=singleton/singleton-policy=foo/election-policy=random:add()
```

Preferences

Additionally, any singleton election policy may indicate a preference for one or more members of a cluster. Preferences may be defined either via node name or via outbound socket binding name. Node preferences always take precedent over the results of an election policy.
e.g.

```
/subsystem=singleton/singleton-policy=foo/election-policy=simple:list-add(name=name-preferences,
value=nodeA)
/subsystem=singleton/singleton-policy=bar/election-policy=random:list-add(name=socket-binding-pref
value=nodeA)
```

Quorum

Network partitions are particularly problematic for singleton services, since they can trigger multiple singleton providers for the same service to run at the same time. To defend against this scenario, a singleton policy may define a quorum that requires a minimum number of nodes to be present before a singleton provider election can take place. A typical deployment scenario uses a quorum of N/2 + 1, where N is the anticipated cluster size. This value can be updated at runtime, and will immediately affect any active singleton services.
e.g.

```
/subsystem=singleton/singleton-policy=foo:write-attribute(name=quorum, value=3)
```

HA environments

The singleton subsystem can be used in a non-HA profile, so long as the cache that it references uses a local-cache configuration. In this manner, an application leveraging singleton functionality (via the singleton API or using a singleton deployment descriptor) will continue function as if the server was a sole member of a cluster. For obvious reasons, the use of a quorum does not make sense in such a configuration.

7.18.2 Singleton deployments

WildFly 10 resurrects the ability to start a given deployment on a single node in the cluster at any given time. If that node shuts down, or fails, the application will automatically start on another node on which the given deployment exists. Long time users of JBoss AS will recognize this functionality as being akin to the HASingletonDeployer, a.k.a. “deploy-hasingleton”, feature of AS6 and earlier.

Usage

A deployment indicates that it should be deployed as a singleton via a deployment descriptor. This can either be a standalone “/META-INF/singleton-deployment.xml” file or embedded within an existing jboss-all.xml descriptor. This descriptor may be applied to any deployment type, e.g. JAR, WAR, EAR, etc., with the exception of a subdeployment within an EAR.
e.g.

```
<singleton-deployment xmlns="urn:jboss:singleton-deployment:1.0" policy="foo"/>
```

The singleton deployment descriptor defines which singleton policy should be used to deploy the application. If undefined, the default singleton policy is used, as defined by the singleton subsystem.

Using a standalone descriptor is often preferable, since it may be overlaid onto an existing deployment archive.
e.g.

```
deployment-overlay add --name=singleton-policy-foo
--content=/META-INF/singleton-deployment.xml=/path/to/singleton-deployment.xml
--deployments=my-app.jar --redeploy-affected
```

7.18.3 Singleton MSC services

WildFly allows any user MSC service to be installed as a singleton MSC service via a public API. Once installed, the service will only ever start on 1 node in the cluster at a time. If the node providing the service is shutdown, or fails, another node on which the service was installed will start automatically.

Installing an MSC service using an existing singleton policy

While singleton MSC services have been around since AS7, WildFly 10 adds the ability to leverage the singleton subsystem to create singleton MSC services from existing singleton policies.

The singleton subsystem exposes capabilities for each singleton policy it defines. These policies, represented via the `org.wildfly.clustering.singleton.SingletonPolicy` interface, can be referenced via the following name: “org.wildfly.clustering.singleton.policy”
e.g.

```
public class MyServiceActivator implements ServiceActivator {
    @Override
    public void activate(ServiceActivatorContext context) {
        ServiceName name = ServiceName.parse(“my.service.name”);
        Service<?> service = new MyService();
        try {
            SingletonPolicy policy = (SingletonPolicy)
context.getServiceRegistry().getRequiredService(ServiceName.parse(SingletonPolicy.CAPABILITY_NAME)
policy.createSingletonServiceBuilder(name, service).build(context.getServiceTarget()).install();
        } catch (InterruptedException e) {
            throw new ServiceRegistryException(e);
        }
    }
}
```

Installing an MSC service using dynamic singleton policy

Alternatively, you can build singleton policy dynamically, which is particularly useful if you want to use a custom singleton election policy. Specifically, `SingletonPolicy` is a generalization of the `org.wildfly.clustering.singleton.SingletonServiceBuilderFactory` interface, which includes support for specifying an election policy and, optionally, a quorum.
e.g.

```
public class MyServiceActivator implements ServiceActivator {
    @Override
    public void activate(ServiceActivatorContext context) {
        String containerName = “server”;
        ElectionPolicy policy = new MySingletonElectionPolicy();
        int quorum = 3;
        ServiceName name = ServiceName.parse(“my.service.name”);
        Service<?> service = new MyService();
        try {
            SingletonServiceBuilderFactory factory = (SingletonServiceBuilderFactory)
context.getServiceRegistry().getRequiredService(SingletonServiceName.BUILDER.getServiceName(contai
factory.createSingletonServiceBuilder(name, service)
                .electionPolicy(policy)
                .quorum(quorum)
                .build(context.getServiceTarget()).install();
        } catch (InterruptedException e) {
            throw new ServiceRegistryException(e);
        }
    }
}
```

7.18.4 Singleton subsystem

WildFly 10 introduces a “singleton” subsystem, which defines a set of policies that define how an HA singleton should behave. A singleton policy can be used to instrument singleton deployments or to create singleton MSC services.

Configuration

The default subsystem configuration from WildFly’s ha and full-ha profile looks like:

```
<subsystem xmlns="urn:jboss:domain:singleton:1.0">
    <singleton-policies default="default">
        <singleton-policy name="default" cache-container="server">
            <simple-election-policy/>
        </singleton-policy>
    </singleton-policies>
</subsystem>
```

A singleton policy defines:

1. A unique name
2. A cache container and cache with which to register singleton provider candidates
3. An election policy
4. A quorum (optional)

One can add a new singleton policy via the following management operation:

```
/subsystem=singleton/singleton-policy=foo:add(cache-container=server)
```

Cache configuration

The cache-container and cache attributes of a singleton policy must reference a valid cache from the Infinispan subsystem. If no specific cache is defined, the default cache of the cache container is assumed. This cache is used as a registry of which nodes can provide a given service and will typically use a replicated-cache configuration.

Election policies

WildFly 10 includes 2 singleton election policy implementations:

- **simple**
  Elects the provider (a.k.a. master) of a singleton service based on a specified position in a circular linked list of eligible nodes sorted by descending age. Position=0, the default value, refers to the oldest node, 1 is second oldest, etc. ; while position=-1 refers to the youngest node, -2 to the second youngest, etc.
  e.g.

```
/subsystem=singleton/singleton-policy=foo/election-policy=simple:add(position=-1)
```

- **random**
  Elects a random member to be the provider of a singleton service
  e.g.

```
/subsystem=singleton/singleton-policy=foo/election-policy=random:add()
```

Preferences

Additionally, any singleton election policy may indicate a preference for one or more members of a cluster. Preferences may be defined either via node name or via outbound socket binding name. Node preferences always take precedent over the results of an election policy.
e.g.

```
/subsystem=singleton/singleton-policy=foo/election-policy=simple:list-add(name=name-preferences,
value=nodeA)
/subsystem=singleton/singleton-policy=bar/election-policy=random:list-add(name=socket-binding-pref
value=nodeA)
```

Quorum

Network partitions are particularly problematic for singleton services, since they can trigger multiple singleton providers for the same service to run at the same time. To defend against this scenario, a singleton policy may define a quorum that requires a minimum number of nodes to be present before a singleton provider election can take place. A typical deployment scenario uses a quorum of N/2 + 1, where N is the anticipated cluster size. This value can be updated at runtime, and will immediately affect any active singleton services.
e.g.

```
/subsystem=singleton/singleton-policy=foo:write-attribute(name=quorum, value=3)
```

HA environments

The singleton subsystem can be used in a non-HA profile, so long as the cache that it references uses a local-cache configuration. In this manner, an application leveraging singleton functionality (via the singleton API or using a singleton deployment descriptor) will continue function as if the server was a sole member of a cluster. For obvious reasons, the use of a quorum does not make sense in such a configuration.

7.18.5 Singleton deployments

WildFly 10 resurrects the ability to start a given deployment on a single node in the cluster at any given time. If that node shuts down, or fails, the application will automatically start on another node on which the given deployment exists. Long time users of JBoss AS will recognize this functionality as being akin to the HASingletonDeployer, a.k.a. “deploy-hasingleton”, feature of AS6 and earlier.

Usage

A deployment indicates that it should be deployed as a singleton via a deployment descriptor. This can either be a standalone “/META-INF/singleton-deployment.xml” file or embedded within an existing jboss-all.xml descriptor. This descriptor may be applied to any deployment type, e.g. JAR, WAR, EAR, etc., with the exception of a subdeployment within an EAR.
e.g.

```
<singleton-deployment xmlns="urn:jboss:singleton-deployment:1.0" policy="foo"/>
```

The singleton deployment descriptor defines which singleton policy should be used to deploy the application. If undefined, the default singleton policy is used, as defined by the singleton subsystem.

Using a standalone descriptor is often preferable, since it may be overlaid onto an existing deployment archive.
e.g.

```
deployment-overlay add --name=singleton-policy-foo
--content=/META-INF/singleton-deployment.xml=/path/to/singleton-deployment.xml
--deployments=my-app.jar --redeploy-affected
```

7.18.6 Singleton MSC services

WildFly allows any user MSC service to be installed as a singleton MSC service via a public API. Once installed, the service will only ever start on 1 node in the cluster at a time. If the node providing the service is shutdown, or fails, another node on which the service was installed will start automatically.

Installing an MSC service using an existing singleton policy

While singleton MSC services have been around since AS7, WildFly 10 adds the ability to leverage the singleton subsystem to create singleton MSC services from existing singleton policies.

The singleton subsystem exposes capabilities for each singleton policy it defines. These policies, represented via the `org.wildfly.clustering.singleton.SingletonPolicy` interface, can be referenced via the following name: “org.wildfly.clustering.singleton.policy”
e.g.

```
public class MyServiceActivator implements ServiceActivator {
    @Override
    public void activate(ServiceActivatorContext context) {
        ServiceName name = ServiceName.parse(“my.service.name”);
        Service<?> service = new MyService();
        try {
            SingletonPolicy policy = (SingletonPolicy)
context.getServiceRegistry().getRequiredService(ServiceName.parse(SingletonPolicy.CAPABILITY_NAME)
policy.createSingletonServiceBuilder(name, service).build(context.getServiceTarget()).install();
        } catch (InterruptedException e) {
            throw new ServiceRegistryException(e);
        }
    }
}
```

Installing an MSC service using dynamic singleton policy

Alternatively, you can build singleton policy dynamically, which is particularly useful if you want to use a custom singleton election policy. Specifically, `SingletonPolicy` is a generalization of the `org.wildfly.clustering.singleton.SingletonServiceBuilderFactory` interface, which includes support for specifying an election policy and, optionally, a quorum.
e.g.

```
public class MyServiceActivator implements ServiceActivator {
    @Override
    public void activate(ServiceActivatorContext context) {
        String containerName = “server”;
        ElectionPolicy policy = new MySingletonElectionPolicy();
        int quorum = 3;
        ServiceName name = ServiceName.parse(“my.service.name”);
        Service<?> service = new MyService();
        try {
            SingletonServiceBuilderFactory factory = (SingletonServiceBuilderFactory)
context.getServiceRegistry().getRequiredService(SingletonServiceName.BUILDER.getServiceName(contai
factory.createSingletonServiceBuilder(name, service)
                .electionPolicy(policy)
                .quorum(quorum)
                .build(context.getServiceTarget()).install();
        } catch (InterruptedException e) {
            throw new ServiceRegistryException(e);
        }
    }
}
```

7.19 Hibernate

7.20 Clustering and Domain Setup Walkthrough

In this article, I'd like to show you how to setup WildFly 9 in domain mode and enable clustering so we could get HA and session replication among the nodes. It's a step to step guide so you can follow the instructions in this article and build the sandbox by yourself 

7.20.1 Preparation & Scenario

Preparation

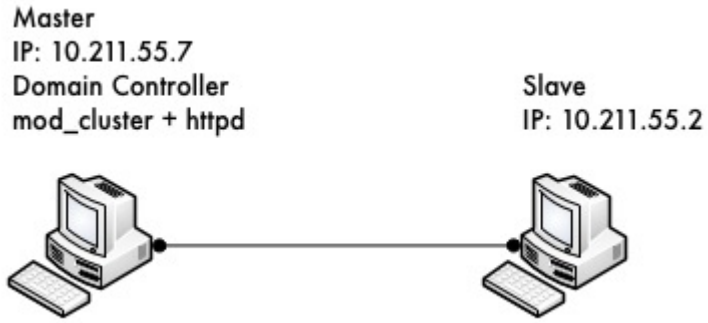
We need to prepare two hosts (or virtual hosts) to do the experiment. We will use these two hosts as following:

- Install Fedora 16 on them (Other linux version may also fine but I'll use Fedora 16 in this article)
- Make sure that they are in same local network
- Make sure that they can access each other via different TCP/UDP ports(better turn off firewall and disable SELinux during the experiment or they will cause network problems).

Scenario

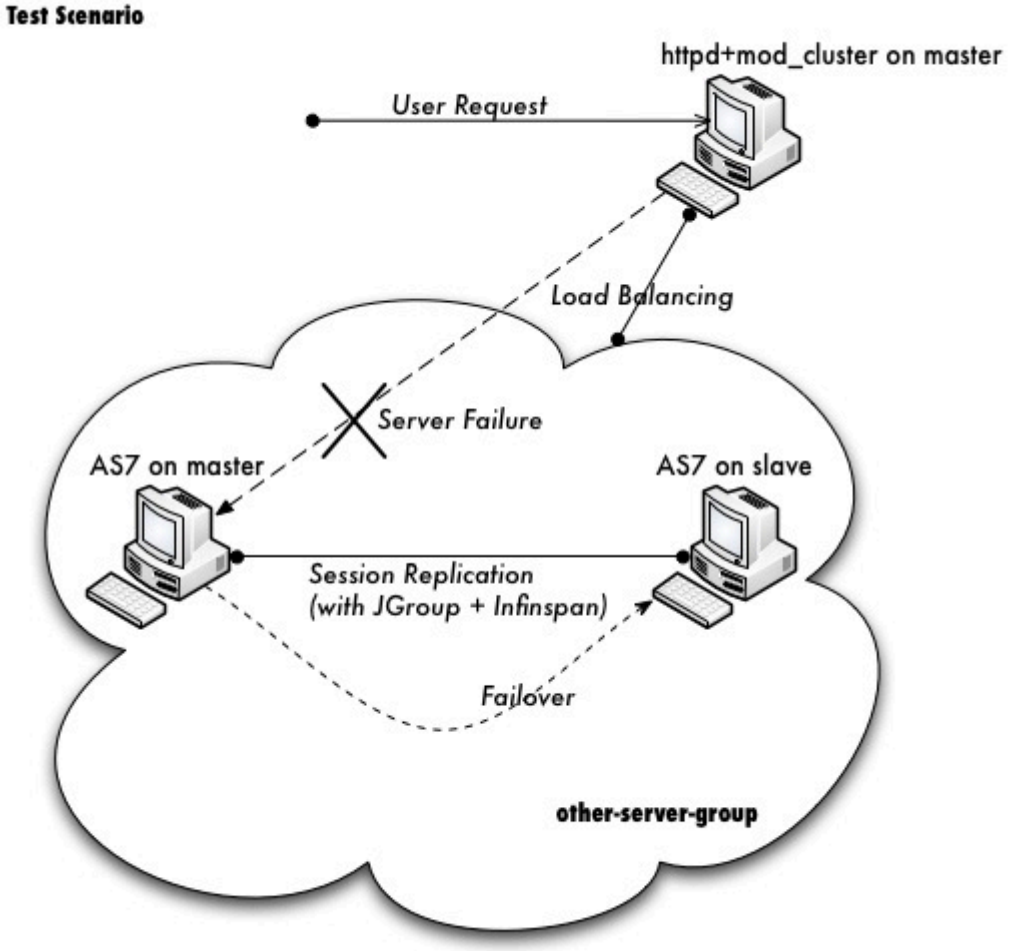
Here are some details on what we are going to do:

- Let's call one host as 'master', the other one as 'slave'.
- Both master and slave will run WildFly 9, and master will run as domain controller, slave will under the domain management of master.
- Apache httpd will be run on master, and in httpd we will enable the mod_cluster module. The WildFly 9 on master and slave will form a cluster and discovered by httpd.



- We will deploy a demo project into domain, and verify that the project is deployed into both master and slave by domain controller. Thus we could see that domain management provide us a single point to manage the deployments across multiple hosts in a single domain.
- We will access the cluster URL and verify that httpd has distributed the request to one of the WildFly host. So we could see the cluster is working properly.

- We will try to make a request on cluster, and if the request is forwarded to master, we then kill the WildFly process on master. After that we will go on requesting cluster and we should see the request is forwarded to slave, but the session is not lost. Our goal is to verify the HA is working and sessions are replicated.
- After previous step finished, we reconnect the master by restarting it. We should see the master is registered back into cluster, also we should see slave sees master as domain controller again and connect to it.



Please don't worry if you cannot digest so many details currently. Let's move on and you will get the points step by step.

7.20.2 Download WildFly 9

First we should download WildFly 9 from the website:

```
http://wildfly.org/downloads/
```

The version I downloaded is WildFly 9.0.0.Final.

After download finished, I got the zip file:

```
wildfly-9.0.0.Final.zip
```

Note: The name of your archive will differ slightly due to version naming conventions.
Then I unzipped the package to master and try to make a test run:

```
unzip wildfly-9.0.0.Final.zip
cd wildfly-9.0.0.Final/bin
./domain.sh
```

If everything ok we should see WildFly successfully startup in domain mode:

```
wildfly-9.0.0.Final/bin$ ./domain.sh
=========================================================================

  JBoss Bootstrap Environment

  JBOSS_HOME: /Users/weli/Downloads/wildfly-9.0.0.Final

  JAVA: /Library/Java/Home/bin/java

  JAVA_OPTS: -Xms64m -Xmx512m -XX:MaxPermSize=256m -Djava.net.preferIPv4Stack=true
-Dorg.jboss.resolver.warning=true -Dsun.rmi.dgc.client.gcInterval=3600000
-Dsun.rmi.dgc.server.gcInterval=3600000 -Djboss.modules.system.pkgs=org.jboss.byteman
-Djava.awt.headless=true

=========================================================================
...

[Server:server-two] 14:46:12,375 INFO  [org.jboss.as] (Controller Boot Thread) JBAS015874:
WildFly 9.0.0.Final "Kenny" started in 8860ms - Started 210 of 258 services (89 services are
lazy, passive or on-demand)
```

Now exit master and let's repeat the same steps on slave host. Finally we get WildFly 9 run on both master and slave, then we could move on to next step.

7.20.3 Domain Configuration

Interface config on master

In this section we'll setup both master and slave for them to run in domain mode. And we will configure master to be the domain controller.

First open the host.xml in master for editing:

```
vi domain/configuration/host.xml
```

The default settings for interface in this file is like:

```
<interfaces>
    <interface name="management">
        <inet-address value="${jboss.bind.address.management:127.0.0.1}"/>
    </interface>
    <interface name="public">
       <inet-address value="${jboss.bind.address:127.0.0.1}"/>
    </interface>
    <interface name="unsecured">
       <inet-address value="127.0.0.1" />
    </interface>
</interfaces>
```

We need to change the address to the management interface so slave could connect to master. The public interface allows the application to be accessed by non-local HTTP, and the unsecured interface allows remote RMI access. My master's ip address is 10.211.55.7, so I change the config to:

```
<interfaces>
    <interface name="management"
        <inet-address value="${jboss.bind.address.management:10.211.55.7}"/>
    </interface>
    <interface name="public">
       <inet-address value="${jboss.bind.address:10.211.55.7}"/>
    </interface>
    <interface name="unsecured">
       <inet-address value="10.211.55.7" />
    </interface>
</interfaces>
```

Interface config on slave

Now we will setup interfaces on slave. Let's edit host.xml. Similar to the steps on master, open host.xml first:

```
vi domain/configuration/host.xml
```

The configuration we'll use on slave is a little bit different, because we need to let slave connect to master. First we need to set the hostname. We change the name property from:

```
<host name="master" xmlns="urn:jboss:domain:3.0">
```

to:

```
<host name="slave" xmlns="urn:jboss:domain:3.0">
```

Then we need to modify domain-controller section so slave can connect to master's management port:

```
<domain-controller>
   <remote protocol="remote" host="10.211.55.7" port="9999" />
</domain-controller>
```

As we know, 10.211.55.7 is the ip address of master.
You may use discovery options to define multiple mechanisms to connect to the remote domain controller :

```
<domain-controller>
 <remote security-realm="ManagementRealm" >
   <discovery-options>
     <static-discovery name="master-native" protocol="remote"  host="10.211.55.7" port=9999" />
     <static-discovery name="master-https" protocol="https-remoting" host="10.211.55.7"
port="9993" security-realm="ManagementRealm"/>
     <static-discovery name="master-http" protocol="http-remoting" host="10.211.55.7"
port="9990" />
   </discovery-options>
        </remote>
    </domain-controller>
```

Finally, we also need to configure interfaces section and expose the management ports to public address:

```
<interfaces>
    <interface name="management">
        <inet-address value="${jboss.bind.address.management:10.211.55.2}"/>
    </interface>
    <interface name="public">
       <inet-address value="${jboss.bind.address:10.211.55.2}"/>
    </interface>
    <interface name="unsecured">
       <inet-address value="10.211.55.2" />
    </interface>
</interfaces>
```

10.211.55.2 is the ip address of the slave. Refer to the domain controller configuration above for an explanation of the management, public, and unsecured interfaces.

It is easier to turn off all firewalls for testing, but in production, you need to enable the firewall and allow access to the following ports: 9999.

Security Configuration

If you start WildFly on both master and slave now, you will see the slave cannot be started with following error:

```
[Host Controller] 20:31:24,575 ERROR [org.jboss.remoting.remote] (Remoting "endpoint" read-1)
JBREM000200: Remote connection failed: javax.security.sasl.SaslException: Authentication failed:
all available authentication mechanisms failed
[Host Controller] 20:31:24,579 WARN  [org.jboss.as.host.controller] (Controller Boot Thread)
JBAS010900: Could not connect to remote domain controller 10.211.55.7:9999
[Host Controller] 20:31:24,582 ERROR [org.jboss.as.host.controller] (Controller Boot Thread)
JBAS010901: Could not connect to master. Aborting. Error was: java.lang.IllegalStateException:
JBAS010942: Unable to connect due to authentication failure.
```

Because we haven't properly set up the authentication between master and slave. Now let's work on it:

Master

In bin directory there is a script called add-user.sh, we'll use it to add new users to the properties file used for domain management authentication:

```
./add-user.sh

Enter the details of the new user to add.
Realm (ManagementRealm) :
Username : admin
Password recommendations are listed below. To modify these restrictions edit the
add-user.properties configuration file.
 - The password should not be one of the following restricted values {root, admin,
administrator}
 - The password should contain at least 8 characters, 1 alphabetic character(s), 1 digit(s), 1
non-alphanumeric symbol(s)
 - The password should be different from the username
Password : passw0rd!
Re-enter Password : passw0rd!
The username 'admin' is easy to guess
Are you sure you want to add user 'admin' yes/no? yes
About to add user 'admin' for realm 'ManagementRealm'
Is this correct yes/no? yes
Added user 'admin' to file
'/home/weli/projs/wildfly-9.0.0.Final/standalone/configuration/mgmt-users.properties'
Added user 'admin' to file
'/home/weli/projs/wildfly-9.0.0.Final/domain/configuration/mgmt-users.properties'
```

As shown above, we have created a user named 'admin' and its password is 'passw0rd!'. Then we add another user called 'slave':

```
./add-user.sh

Enter the details of the new user to add.
Realm (ManagementRealm) :
Username : slave
Password recommendations are listed below. To modify these restrictions edit the
add-user.properties configuration file.
 - The password should not be one of the following restricted values {root, admin,
administrator}
 - The password should contain at least 8 characters, 1 alphabetic character(s), 1 digit(s), 1
non-alphanumeric symbol(s)
 - The password should be different from the username
Password : passw0rd!
Re-enter Password : passw0rd!
What groups do you want this user to belong to? (Please enter a comma separated list, or leave
blank for none)[  ]:
About to add user 'slave' for realm 'ManagementRealm'
Is this correct yes/no? yes
Added user 'slave' to file
'/home/weli/projs/wildfly-9.0.0.Final/standalone/configuration/mgmt-users.properties'
Added user 'slave' to file
'/home/weli/projs/wildfly-9.0.0.Final/domain/configuration/mgmt-users.properties'
Added user 'slave' with groups  to file
'/home/weli/projs/wildfly-9.0.0.Final/standalone/configuration/mgmt-groups.properties'
Added user 'slave' with groups  to file
'/home/weli/projs/wildfly-9.0.0.Final/domain/configuration/mgmt-groups.properties'
Is this new user going to be used for one AS process to connect to another AS process?
e.g. for a slave host controller connecting to the master or for a Remoting connection for
server to server EJB calls.
yes/no? yes
To represent the user add the following to the server-identities definition <secret
value="cGFzc3cwcmQh" />
```

We will use this user for slave host to connect to master. The add-user.sh will let you choose the type of the user. Here we need to choose 'Management User' type for both 'admin' and 'slave' account.

Slave

In slave we need to configure host.xml for authentication. We should change the security-realms section as following:

```
<security-realms>
   <security-realm name="ManagementRealm">
       <server-identities>
           <secret value="cGFzc3cwcmQh" />
           <!-- This is required for SSL remoting -->
           <ssl>
             <keystore path="server.keystore" relative-to="jboss.domain.config.dir"
keystore-password="jbossas" alias="jboss" key-password="jbossas"/>
           </ssl>
       </server-identities>
       <authentication>
           <properties path="mgmt-users.properties" relative-to="jboss.domain.config.dir"/>
       </authentication>
   </security-realm>
</security-realms>
```

We've added server-identities into security-realm, which is used for authentication host when slave tries to connect to master. In secret value property we have 'cGFzc3cwcmQh', which is the base64 code for 'passw0rd!'. You can generate this value by using a base64 calculator such as the one at http://www.webutils.pl/index.php?idx=base64.

Then in domain controller section we also need to add security-realm property:

```
<domain-controller>
   <remote protocol="remote" host="10.211.55.7" port="9999" username="slave"
security-realm="ManagementRealm"  />
</domain-controller>
```

So the slave host could use the authentication information we provided in 'ManagementRealm'.

Dry Run

Now everything is set for the two hosts to run in domain mode. Let's start them by running domain.sh on both hosts. If everything goes fine, we could see from the log on master:

```
[Host Controller] 21:30:52,042 INFO  [org.jboss.as.domain] (management-handler-threads - 1)
JBAS010918: Registered remote slave host slave
```

That means all the configurations are correct and two hosts are run in domain mode now as expected. Hurrah!

7.20.4 Deployment

Now we can deploy a demo project into the domain. I have created a simple project located at:

```
https://github.com/liweinan/cluster-demo
```

We can use git command to fetch a copy of the demo:

```
git clone git://github.com/liweinan/cluster-demo.git
```

In this demo project we have a very simple web application. In web.xml we've enabled session replication by adding following entry:

```
<distributable/>
```

And it contains a jsp page called put.jsp which will put current time to a session entry called 'current.time':

```
<%
    session.setAttribute("current.time", new java.util.Date());
%>
```

Then we could fetch this value from get.jsp:

```
The time is <%= session.getAttribute("current.time") %>
```

It's an extremely simple project but it could help us to test the cluster later: We will access put.jsp from cluster and see the request are distributed to master, then we disconnect master and access get.jsp. We should see the request is forwarded to slave but the 'current.time' value is held by session replication. We'll cover more details on this one later.

Let's go back to this demo project. Now we need to create a war from it. In the project directory, run the following command to get the war:

```
mvn package
```

It will generate cluster-demo.war. Then we need to deploy the war into domain. First we should access the http management console on master(Because master is acting as domain controller):

```
http://10.211.55.7:9990
```

It will popup a windows ask you to input account name and password, we can use the 'admin' account we've added just now. After logging in we could see the 'Server Instances' window. By default there are three servers listed, which are:

- server-one
- server-two
- server-three

We could see server-one and server-two are in running status and they belong to main-server-group; server-three is in idle status, and it belongs to other-server-group.

All these servers and server groups are set in domain.xml on master as7. What we are interested in is the 'other-server-group' in domain.xml:

```
<server-group name="other-server-group" profile="ha">
   <jvm name="default">
       <heap size="64m" max-size="512m"/>
   </jvm>
   <socket-binding-group ref="ha-sockets"/>
</server-group>
```

We could see this server-group is using 'ha' profile, which then uses 'ha-sockets' socket binding group. It enable all the modules we need to establish cluster later(including infinispan, jgroup and mod_cluster modules). So we will deploy our demo project into a server that belongs to 'other-server-group', so 'server-three' is our choice.

In newer version of WildFly, the profile 'ha' changes to 'full-ha':

```
<server-group name="other-server-group" profile="full-ha">
```

Let's go back to domain controller's management console:

```
http://10.211.55.7:9990
```

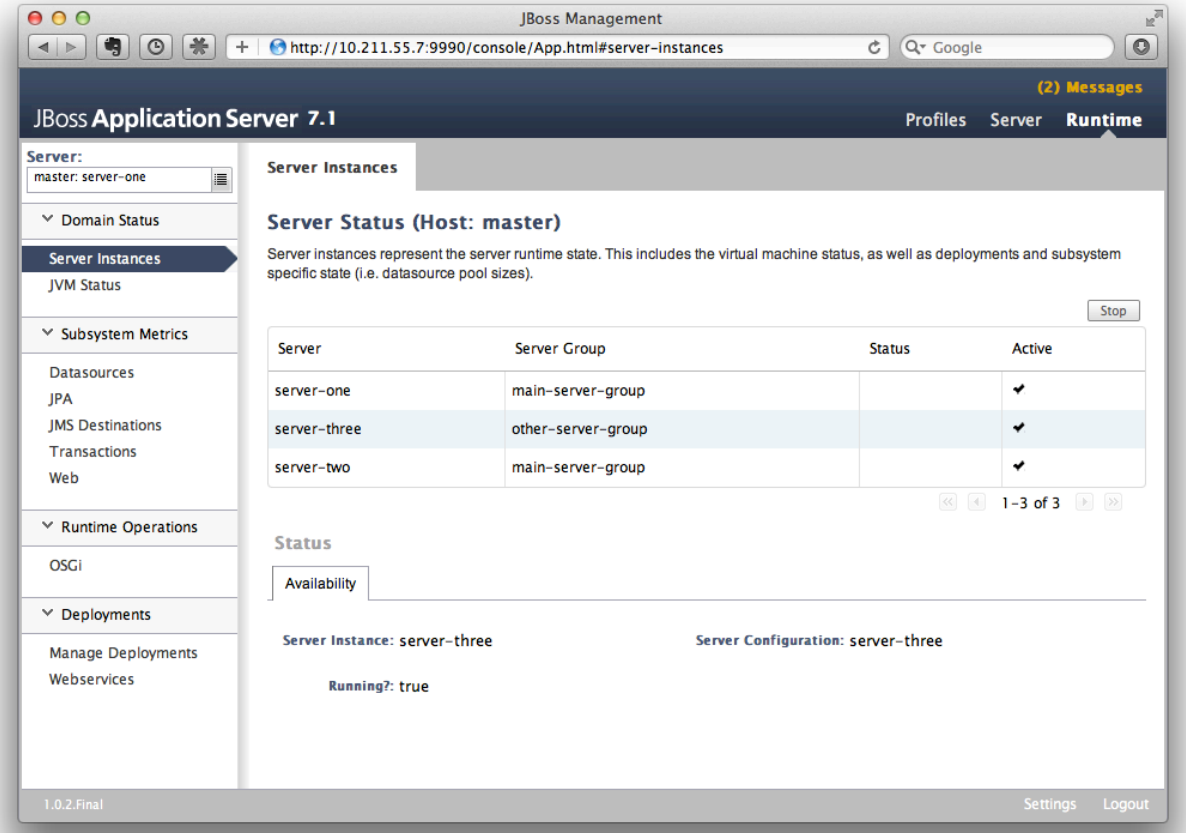
Now server-three is not running, so let's click on 'server-three' and then click the 'start' button at bottom right of the server list. Wait a moment and server-three should start now.

Now we should also enable 'server-three' on slave: From the top of menu list on left side of the page, we could see now we are managing master currently. Click on the list, and click 'slave', then choose 'server-three', and we are in slave host management page now.

Then repeat the steps we've done on master to start 'server-three' on slave.

server-three on master and slave are two different hosts, their names can be different.

After server-three on both master and slave are started, we will add our cluster-demo.war for deployment. Click on the 'Manage Deployments' link at the bottom of left menu list.

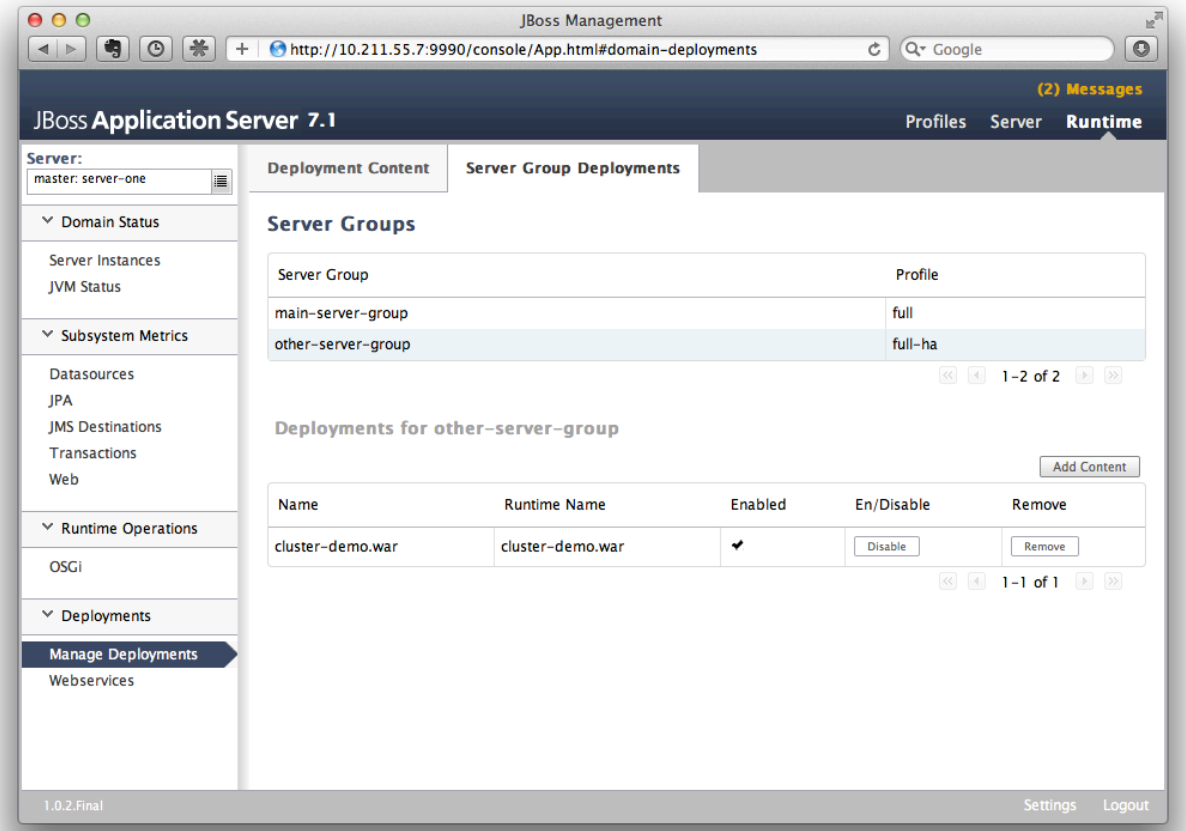


(We should ensure the server-three should be started on both master and slave)

After enter 'Manage Deployments' page, click 'Add Content' at top right corner. Then we should choose our cluster-demo.war, and follow the instruction to add it into our content repository.

Now we can see cluster-demo.war is added. Next we click 'Add to Groups' button and add the war to 'other-server-group' and then click 'save'.

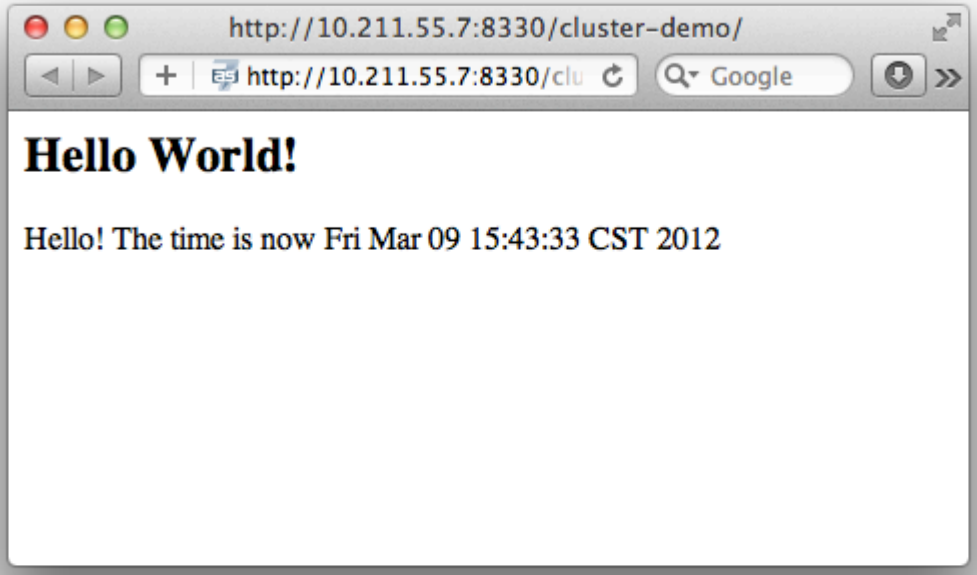
Wait a few seconds, management console will tell you that the project is deployed into 'other-server-group'.



Please note we have two hosts participate in this server group, so the project should be deployed in both master and slave now - that's the power of domain management.

Now let's verify this, trying to access cluster-demo from both master and slave, and they should all work now:

```
http://10.211.55.7:8330/cluster-demo/
```




```
http://10.211.55.2:8330/cluster-demo/
```

Now that we have finished the project deployment and see the usages of domain controller, we will then head up for using these two hosts to establish a cluster

Why is the port number 8330 instead of 8080? Please check the settings in host.xml on both master and slave:

```
<server name="server-three" group="other-server-group" auto-start="false">
    <!-- server-three avoids port conflicts by incrementing the ports in
         the default socket-group declared in the server-group -->
    <socket-bindings port-offset="250"/>
</server>
```

The port-offset is set to 250, so 8080 + 250 = 8330

Now we quit the WildFly process on both master and slave. We have some work left on host.xml configurations. Open the host.xml of master, and do some modifications the servers section from:

```
<server name="server-three" group="other-server-group" auto-start="false">
    <!-- server-three avoids port conflicts by incrementing the ports in
         the default socket-group declared in the server-group -->
    <socket-bindings port-offset="250"/>
</server>
```

to:

```
<server name="server-three" group="other-server-group" auto-start="true">
    <!-- server-three avoids port conflicts by incrementing the ports in
         the default socket-group declared in the server-group -->
    <socket-bindings port-offset="250"/>
</server>
```

We've set auto-start to true so we don't need to enable it in management console each time WildFly restart. Now open slave's host.xml, and modify the server-three section:

```
<server name="server-three-slave" group="other-server-group" auto-start="true">
    <!-- server-three avoids port conflicts by incrementing the ports in
         the default socket-group declared in the server-group -->
    <socket-bindings port-offset="250"/>
</server>
```

Besides setting auto-start to true, we've renamed the 'server-three' to 'server-three-slave'. We need to do this because mod_cluster will fail to register the hosts with same name in a single server group. It will cause name conflict.

After finishing the above configuration, let's restart two as7 hosts and go on cluster configuration.

7.20.5 Cluster Configuration

We will use mod_cluster + apache httpd on master as our cluster controller here. Because WildFly 8 has been configured to support mod_cluster out of box so it's the easiest way.

The WildFly 8 domain controller and httpd are not necessary to be on same host. But in this article I just install them all on master for convenience.

First we need to ensure that httpd is installed:

```
sudo yum install httpd
```

And then we need to download newer version of mod_cluster from its website:

```
http://www.jboss.org/mod_cluster/downloads
```

The version I downloaded is:

```
http://downloads.jboss.org/mod_cluster/1.1.3.Final/mod_cluster-1.1.3.Final-linux2-x86-so.tar.gz
```

Jean-Frederic has suggested to use mod_cluster 1.2.x. Because 1.1.x it is affected by CVE-2011-4608

With mod_cluster-1.2.0 you need to add EnableMCPMReceive in the VirtualHost.

Then we extract it into:

```
/etc/httpd/modules
```

Then we edit httpd.conf:

```
sudo vi /etc/httpd/conf/httpd.conf
```

We should add the modules:

```
LoadModule slotmem_module modules/mod_slotmem.so
LoadModule manager_module modules/mod_manager.so
LoadModule proxy_cluster_module modules/mod_proxy_cluster.so
LoadModule advertise_module modules/mod_advertise.so
```

Please note we should comment out:

```
#LoadModule proxy_balancer_module modules/mod_proxy_balancer.so
```

This is conflicted with cluster module. And then we need to make httpd to listen to public address so we could do the testing. Because we installed httpd on master host so we know the ip address of it:

```
Listen 10.211.55.7:80
```

Then we do the necessary configuration at the bottom of httpd.conf:

```
# This Listen port is for the mod_cluster-manager, where you can see the status of mod_cluster.
# Port 10001 is not a reserved port, so this prevents problems with SELinux.
Listen 10.211.55.7:10001
# This directive only applies to Red Hat Enterprise Linux. It prevents the temmporary
# files from being written to /etc/httpd/logs/ which is not an appropriate location.
MemManagerFile /var/cache/httpd

<VirtualHost 10.211.55.7:10001>

  <Directory />
    Order deny,allow
    Deny from all
    Allow from 10.211.55.
  </Directory>


  # This directive allows you to view mod_cluster status at URL
http://10.211.55.4:10001/mod_cluster-manager
  <Location /mod_cluster-manager>
   SetHandler mod_cluster-manager
   Order deny,allow
   Deny from all
   Allow from 10.211.55.
  </Location>

  KeepAliveTimeout 60
  MaxKeepAliveRequests 0

  ManagerBalancerName other-server-group
  AdvertiseFrequency 5

</VirtualHost>
```

For more details on mod_cluster configurations please see this document:

```
http://docs.jboss.org/mod_cluster/1.1.0/html/Quick_Start_Guide.html
```

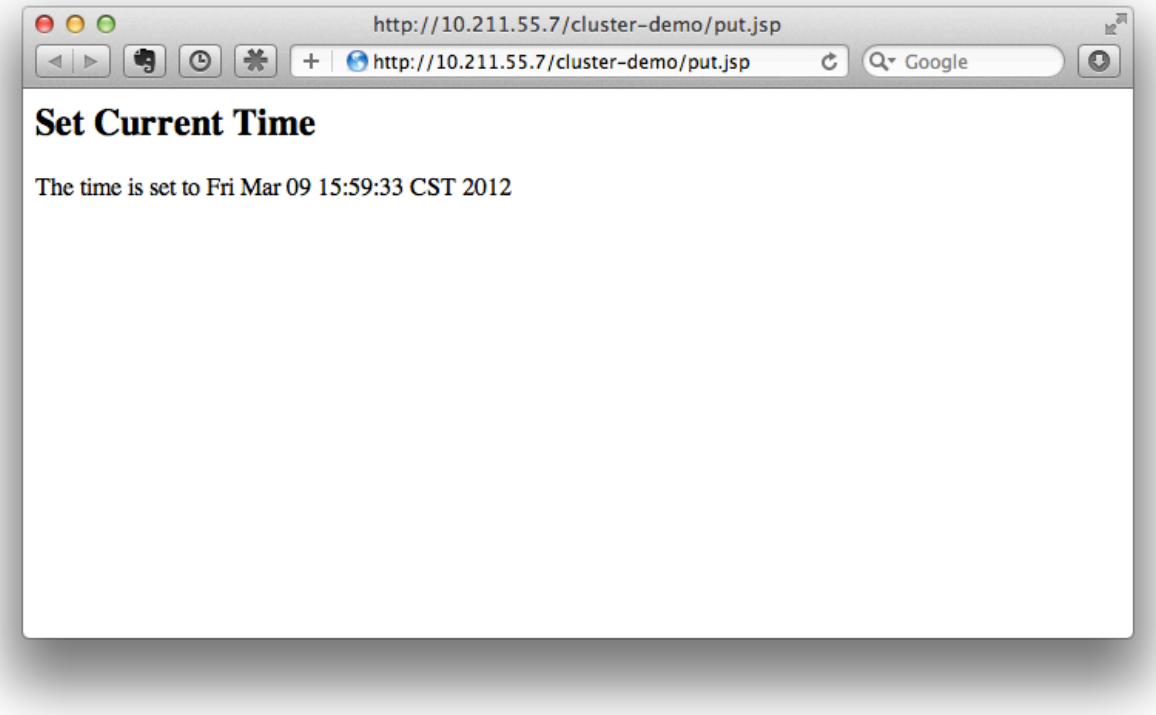
7.20.6 Testing

If everything goes fine we can start httpd service now:

```
service httpd start
```

Now we access the cluster:

```
http://10.211.55.7/cluster-demo/put.jsp
```



We should see the request is distributed to one of the hosts(master or slave) from the WildFly log. For me the request is sent to master:

```
[Server:server-three] 16:06:22,256 INFO  [stdout] (http-10.211.55.7-10.211.55.7-8330-4) Putting
date now
```

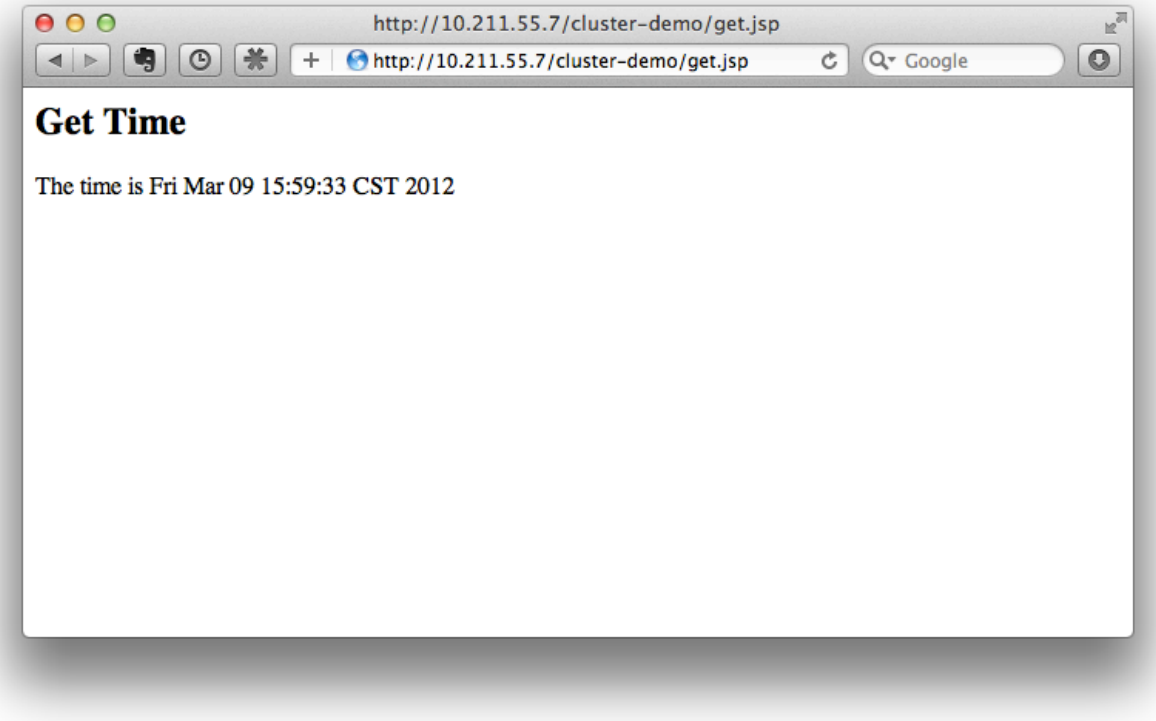
Now I disconnect master by using the management interface. Select 'runtime' and the server 'master' in the upper corners.

Select 'server-three' and kick the stop button, the active-icon should change.

Killing the server by using system commands will have the effect that the Host-Controller restart the instance imediately!

Then wait for a few seconds and access cluster:

```
http://10.211.55.7/cluster-demo/get.jsp
```



Now the request should be served by slave and we should see the log from slave:

```
[Server:server-three-slave] 16:08:29,860 INFO  [stdout] (http-10.211.55.2-10.211.55.2-8330-1)
Getting date now
```

And from the get.jsp we should see that the time we get is the same we've put by 'put.jsp'. Thus it's proven that the session is correctly replicated to slave.

Now we restart master and should see the host is registered back to cluster.

It doesn't matter if you found the request is distributed to slave at first time. Then just disconnect slave and do the testing, the request should be sent to master instead. The point is we should see the request is redirect from one host to another and the session is held.

7.20.7 Special Thanks

Wolf-Dieter Fink has contributed the updated add-user.sh usages and configs in host.xml from 7.1.0.Final.
Jean-Frederic Clere provided the mod_cluster 1.2.0 usages.
Misty Stanley-Jones has given a lot of suggestions and helps to make this document readable.

7.21 Changes From Previous Versions

Describe here key changes between releases.

7.21.1 Key changes

7.21.2 Migration to Wildfly

7.22 Related Topics

This section describes additional issues related to the clustering subsystems.

7.22.1 Modularity And Class Loading

Describe classloading and monitoring framework as it affects clustering applications.

7.22.2 Monitoring

Describe resources available for monitoring clustered applications.

8 Getting Started Developing Applications Guide

This guide has moved to https://github.com/wildfly/quickstart/blob/10.x/guide/Introduction.asciidoc

8.1 Introduction

This page has moved to https://github.com/wildfly/quickstart/blob/10.x/guide/Introduction.asciidoc

8.2 Getting started with WildFly

This page has moved to https://github.com/wildfly/quickstart/blob/10.x/guide/GettingStarted.asciidoc

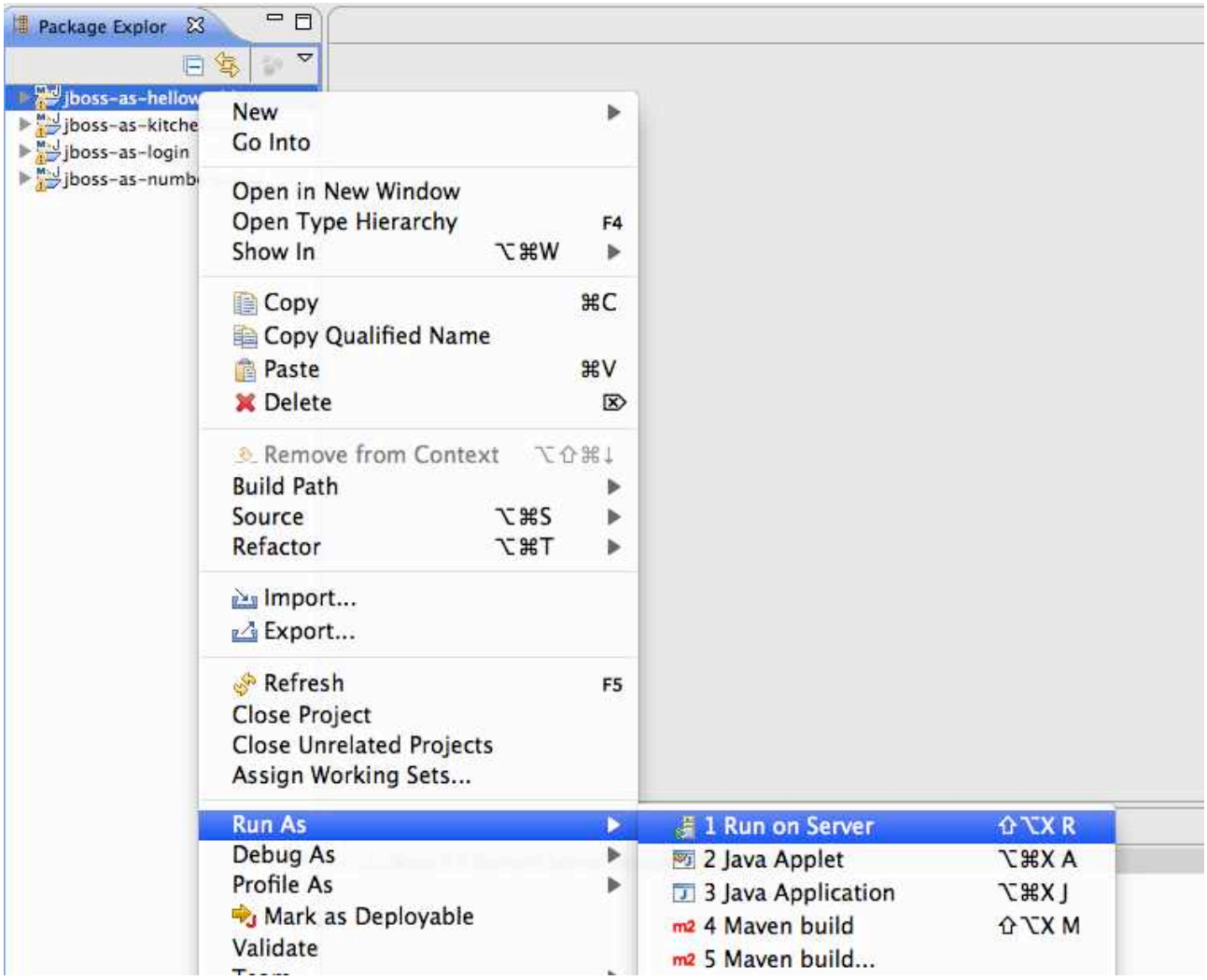
8.3 Helloworld quickstart

This page has moved to https://github.com/wildfly/quickstart/blob/10.x/guide/HelloworldQuickstart.asciidoc

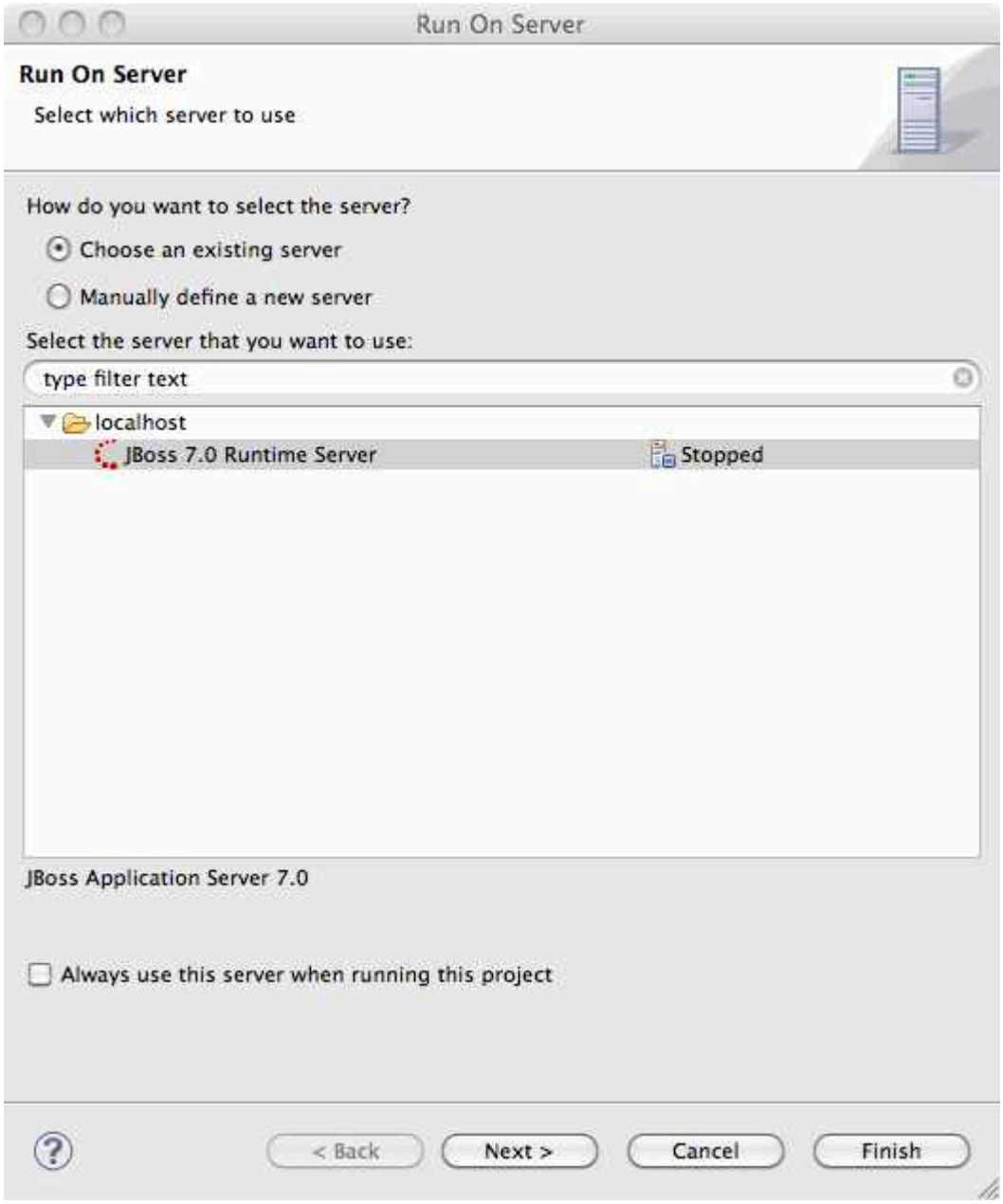
8.3.1 Deploying the Helloworld example using Eclipse

You may choose to deploy the example using Eclipse. You'll need to have JBoss AS started in Eclipse (as described in [Starting JBoss AS from Eclipse with JBoss Tools]) and to have imported the quickstarts into Eclipse (as described in [Importing the quickstarts into Eclipse]).

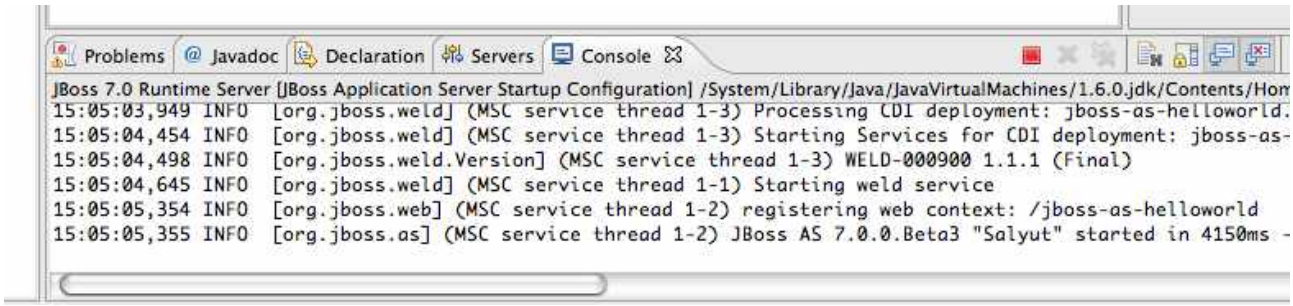
With the quickstarts imported, you can deploy the example by right clicking on the `jboss-as-helloworld` project, and choosing `Run As -> Run On Server`:



Make sure the JBoss AS server is selected, and hit `Finish`:



You should see JBoss AS start up (unless you already started it in [Starting JBoss AS from Eclipse with JBoss Tools]) and the application deploy in the Console log:



8.3.2 The helloworld example in depth

This page has moved to
http://www.jboss.org/jdf/quickstarts/jboss-as-quickstart/guide/HelloworldQuickstart/#_the_helloworld_quicksta

8.4 Numberguess quickstart

This page has moved to
https://github.com/wildfly/quickstart/blob/10.x/guide/NumberguessQuickstart.asciidoc

8.4.1 Deploying the Numberguess example using Eclipse

This page has moved to
http://www.jboss.org/jdf/quickstarts/jboss-as-quickstart/guide/NumberguessQuickstart/#_deploying_the_numb

8.4.2 The numberguess example in depth

This page has moved to
http://www.jboss.org/jdf/quickstarts/jboss-as-quickstart/guide/NumberguessQuickstart/#_the_numberguess_q

8.5 Greeter quickstart

This page has moved to https://github.com/wildfly/quickstart/blob/10.x/guide/GreeterQuickstart.asciidoc

8.5.1 Deploying the Login example using Eclipse

This page has moved to http://www.jboss.org/jdf/quickstarts/jboss-as-quickstart/guide/GreeterQuickstart/

8.5.2 The login example in depth

This page has moved to
http://www.jboss.org/jdf/quickstarts/jboss-as-quickstart/guide/GreeterQuickstart/#greeter_in_depth

8.6 Kitchensink quickstart

This page has moved to https://github.com/wildfly/quickstart/blob/10.x/guide/KitchensinkQuickstart.asciidoc

8.6.1 Deploying the Kitchensink example using Eclipse

This page has moved to
http://www.jboss.org/jdf/quickstarts/jboss-as-quickstart/guide/KitchensinkQuickstart/#_deploying_the_kitchens

8.6.2 The kitchensink example in depth

This page has moved to
http://www.jboss.org/jdf/quickstarts/jboss-as-quickstart/guide/KitchensinkQuickstart/#_the_kitchensink_quicks

8.7 Creating your own application

This page has moved to https://github.com/wildfly/quickstart/blob/10.x/guide/Archetype.asciidoc

8.7.1 Creating your own application using Eclipse

This page has moved to http://www.jboss.org/jdf/quickstarts/jboss-as-quickstart/guide/Archetype/

8.8 More Resources

Getting Started Guide	The Getting Started Guide covers topics such as server layout (what you can configure where), data source definition, and using the web management interface.
Torquebox	Torque Box allows you to use all the familiar services from JBoss AS 7, but with Ruby.
JBoss AS 7 FAQ	Frequently Asked Questions for JBoss AS 7

8.8.1 Developing JSF Project Using JBoss AS7, Maven and IntelliJ

JBoss AS7 is a very 'modern' application server that has very fast startup speed. So it's an excellent container to test your JSF project. In this article, I'd like to show you how to use AS7, maven and IntelliJ together to develop your JSF project.

In this article I'd like to introduce the following things:

- Create a project using Maven
- Add JSF into project
- Writing Code
- Add JBoss AS 7 deploy plugin into project
- Deploy project to JBoss AS 7
- Import project into IntelliJ
- Add IntelliJ JSF support to project
- Add JBoss AS7 to IntelliJ
- Debugging project with IntelliJ and AS7

I won't explain many basic concepts about AS7, maven and IntelliJ in this article because there are already many good introductions on these topics. So before doing the real work, there some preparations should be done firstly:

Download JBoss AS7

It could be downloaded from here: http://www.jboss.org/jbossas/downloads/

Using the latest release would be fine. When I'm writing this article the latest version is 7.1.1.Final.

Install Maven

Please make sure you have maven installed on your machine. Here is my environment:

```
weli@power:~$ mvn -version
Apache Maven 3.0.3 (r1075438; 2011-03-01 01:31:09+0800)
Maven home: /usr/share/maven
Java version: 1.6.0_33, vendor: Apple Inc.
Java home: /System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home
Default locale: en_US, platform encoding: MacRoman
OS name: "mac os x", version: "10.8", arch: "x86_64", family: "mac"
```

Get IntelliJ

In this article I'd like to use IntelliJ Ultimate Edition as the IDE for development, it's a commercial software and can be downloaded from: http://www.jetbrains.com/idea/

The version I'm using is IntelliJ IDEA Ultimate 11.1

After all of these prepared, we can dive into the real work:

Create a project using Maven

Use the following maven command to create a web project:

```
mvn archetype:create -DarchetypeGroupId=org.apache.maven.archetypes \
-DarchetypeArtifactId=maven-archetype-webapp \
-DarchetypeVersion=1.0 \
-DgroupId=net.bluedash \
-DartifactId=jsfdemo \
-Dversion=1.0-SNAPSHOT
```

If everything goes fine maven will generate the project for us:

The contents of the project is shown as above.

Add JSF into project

The JSF library is now included in maven repo, so we can let maven to manage the download for us. First is to add repository into our pom.xml:

```
<repository>
  <id>jvnet-nexus-releases</id>
  <name>jvnet-nexus-releases</name>
  <url>https://maven.java.net/content/repositories/releases/</url>
</repository>
```

Then we add JSF dependency into pom.xml:

```
<dependency>
    <groupId>javax.faces</groupId>
    <artifactId>jsf-api</artifactId>
    <version>2.1</version>
    <scope>provided</scope>
</dependency>
```

Please note the 'scope' is 'provided', because we don't want to bundle the jsf.jar into the war produced by our project later, as JBoss AS7 already have jsf bundled in.

Then we run 'mvn install' to update the project, and maven will download jsf-api for us automatically.

Writing Code

Writing JSF code in this article is trivial, so I've put written a project called 'jsfdemo' onto github:

https://github.com/liweinan/jsfdemo

Please clone this project into your local machine, and import it into IntelliJ following the steps described as above.

Add JBoss AS 7 deploy plugin into project

JBoss AS7 has provide a set of convenient maven plugins to perform daily tasks such as deploying project into AS7. In this step let's see how to use it in our project.

We should put AS7's repository into pom.xml:

```
<repository>
    <id>jboss-public-repository-group</id>
    <name>JBoss Public Repository Group</name>
    <url>http://repository.jboss.org/nexus/content/groups/public/</url>
    <layout>default</layout>
    <releases>
        <enabled>true</enabled>
        <updatePolicy>never</updatePolicy>
    </releases>
    <snapshots>
        <enabled>true</enabled>
        <updatePolicy>never</updatePolicy>
    </snapshots>
</repository>
```

And also the plugin repository:

```
<pluginRepository>
    <id>jboss-public-repository-group</id>
    <name>JBoss Public Repository Group</name>
    <url>http://repository.jboss.org/nexus/content/groups/public/</url>
    <releases>
        <enabled>true</enabled>
    </releases>
    <snapshots>
        <enabled>true</enabled>
    </snapshots>
</pluginRepository>
```

And put jboss deploy plugin into 'build' section:

```
<plugin>
    <groupId>org.jboss.as.plugins</groupId>
    <artifactId>jboss-as-maven-plugin</artifactId>
    <executions>
        <execution>
            <phase>package</phase>
            <goals>
                <goal>deploy</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

I've put the final version pom.xml here to check whether your modification is correct:

https://github.com/liweinan/jsfdemo/blob/master/pom.xml

Now we have finished the setup work for maven.

Deploy project to JBoss AS 7

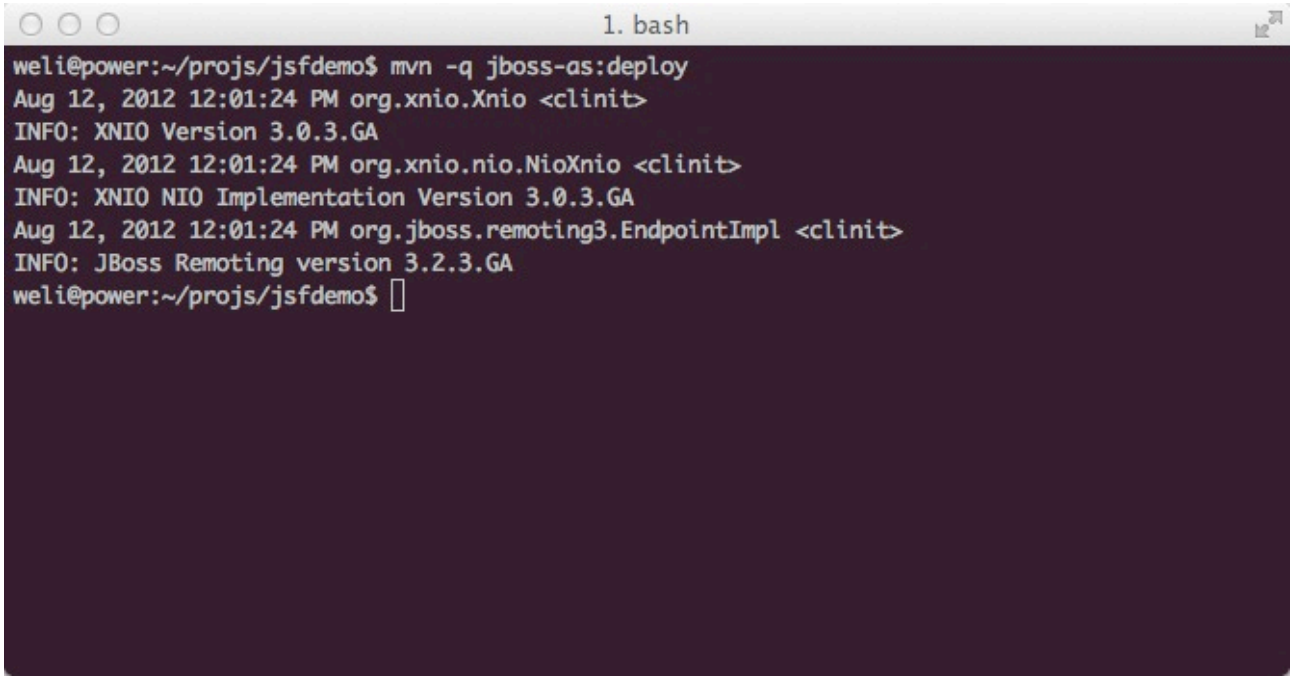
To deploy the project to JBoss AS7, we should start AS7 firstly. In JBoss AS7 directory, run following command:

```
bin/standalone.sh
```

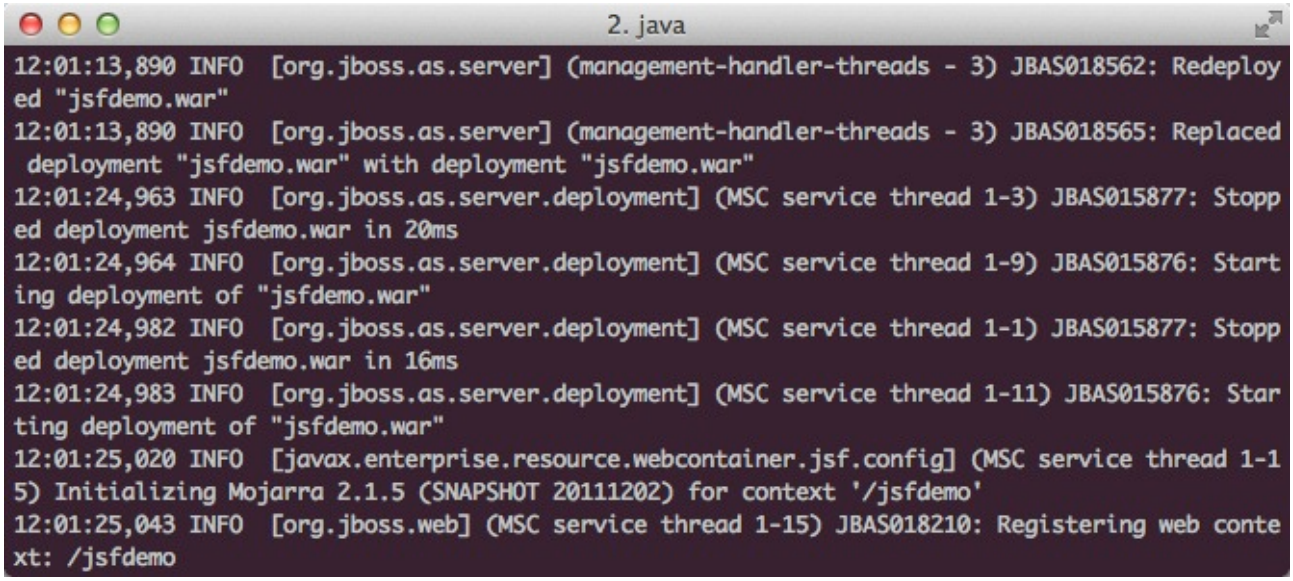
AS7 should start in a short time. Then let's go back to our project directory and run maven command:

```
mvn -q jboss-as:deploy
```

Maven will use some time to download necessary components for a while, so please wait patiently. After a while, we can see the result:



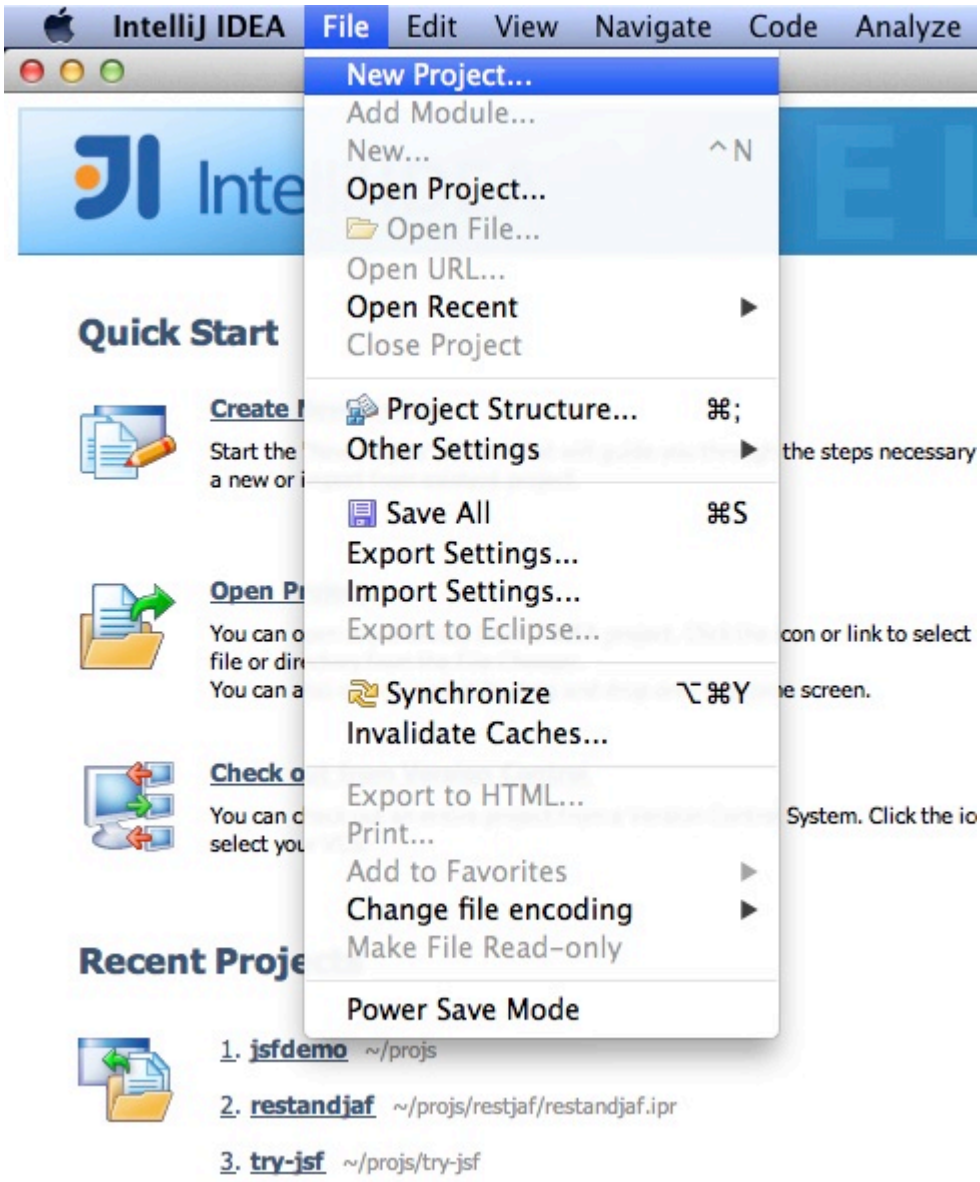
And if you check the console output of AS7, you can see the project is deployed:



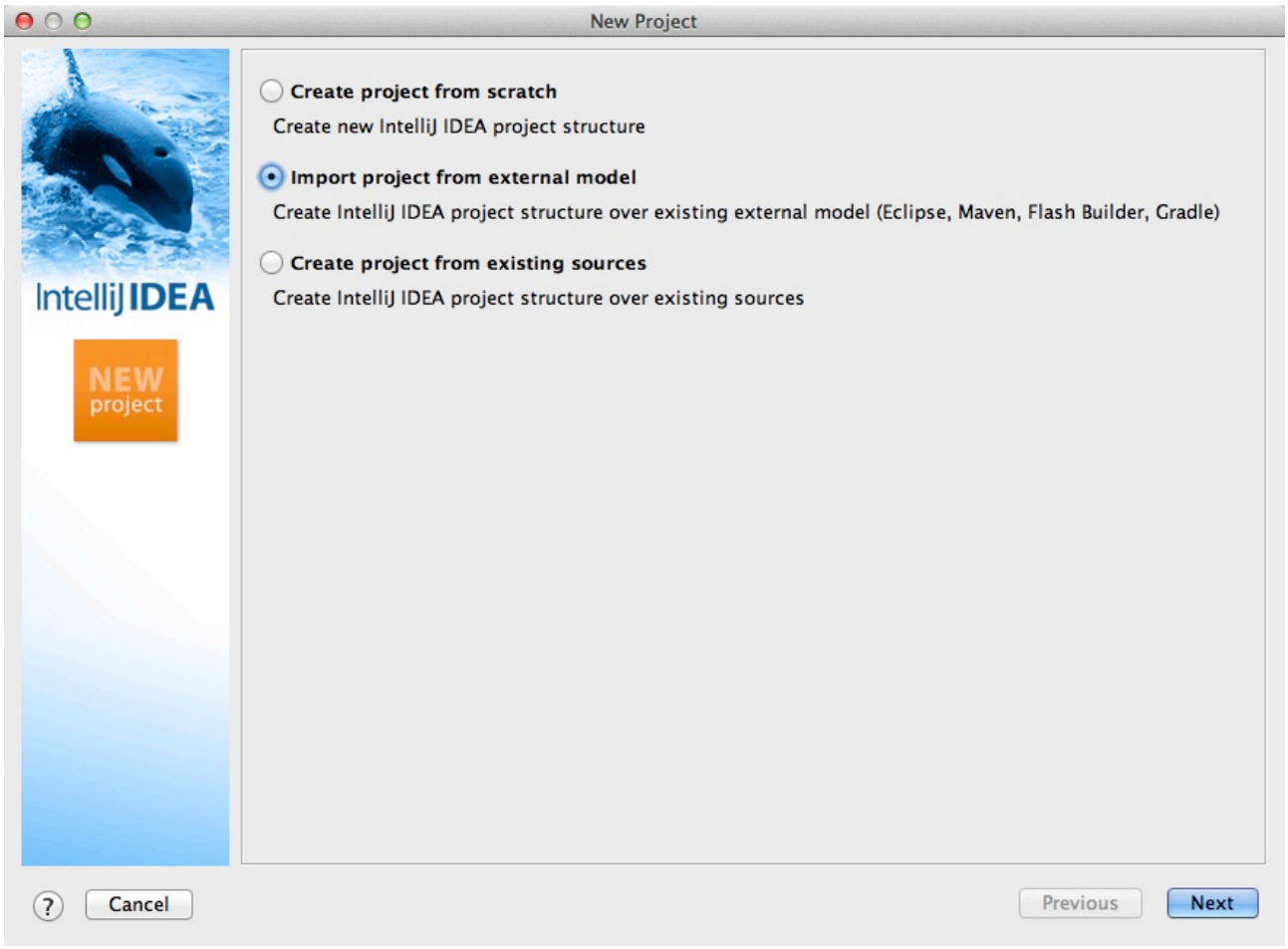
Now we have learnt how to create a JSF project and deploy it to AS7 without any help from graphical tools. Next let's see how to use IntelliJ IDEA to go on developing/debugging our project.

Import project into IntelliJ

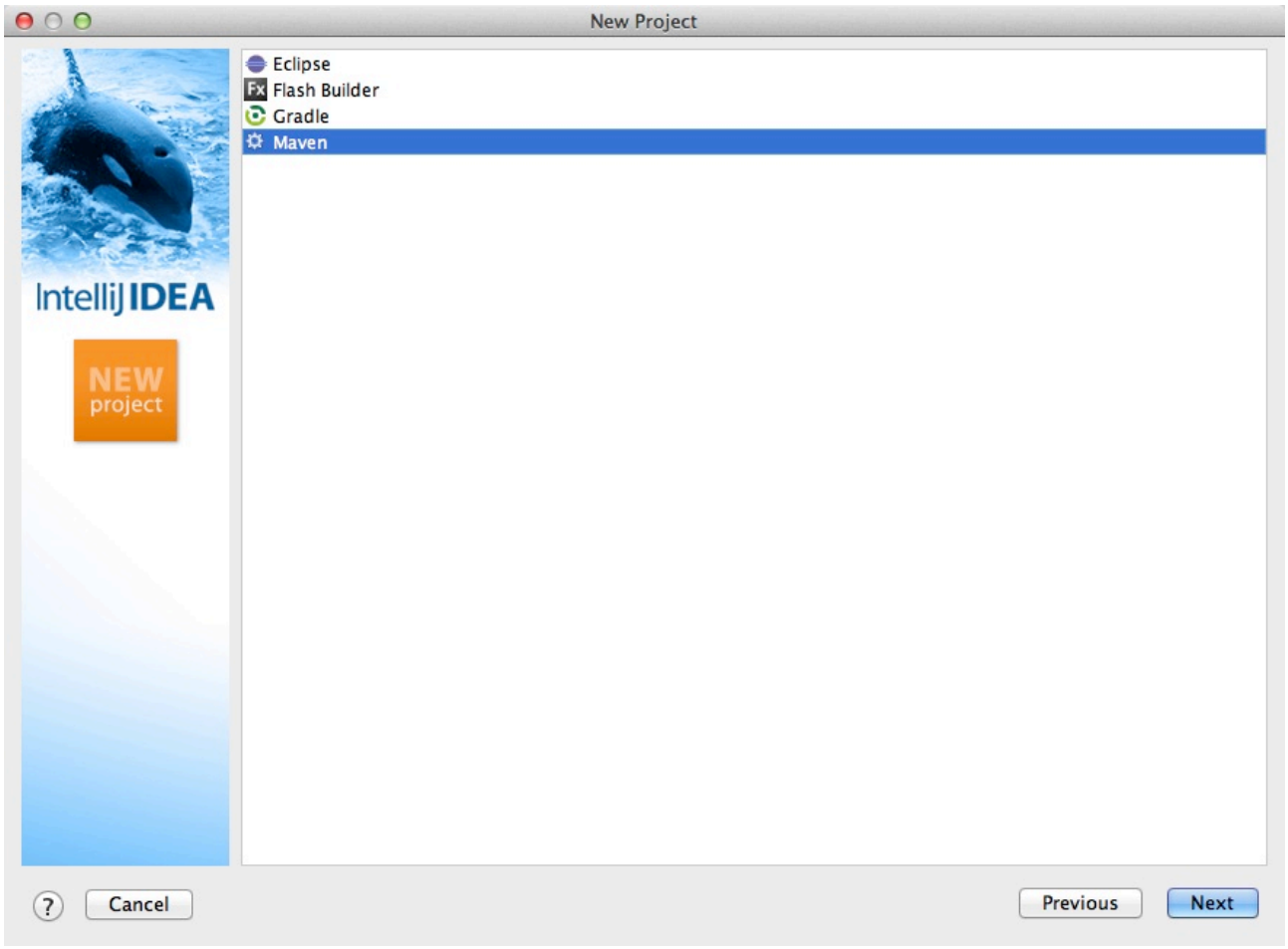
Now it's time to import the project into IntelliJ. Now let's open IntelliJ, and choose 'New Project...':



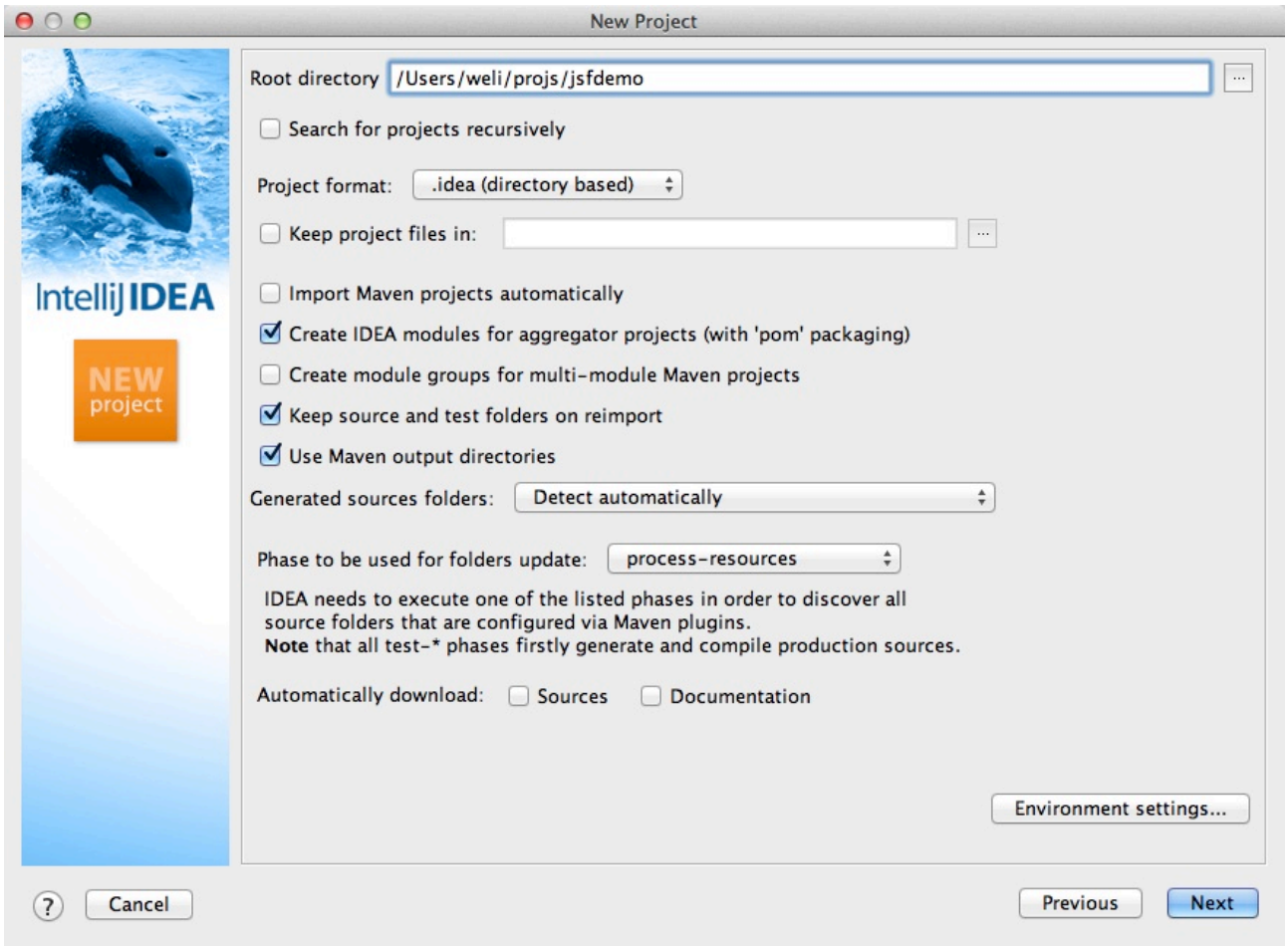
The we choose 'Import project from external model':



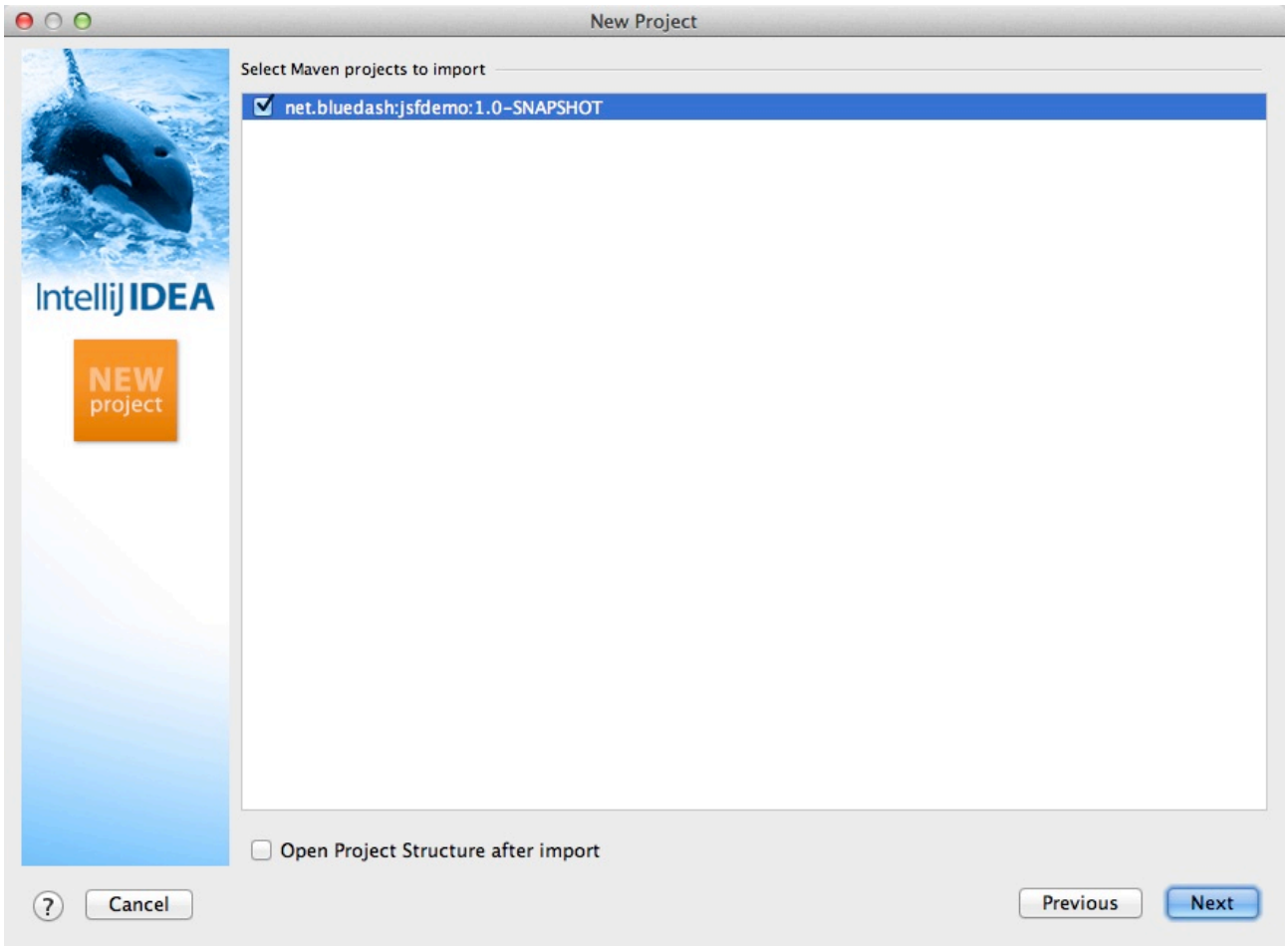
Next step is choosing 'Maven':



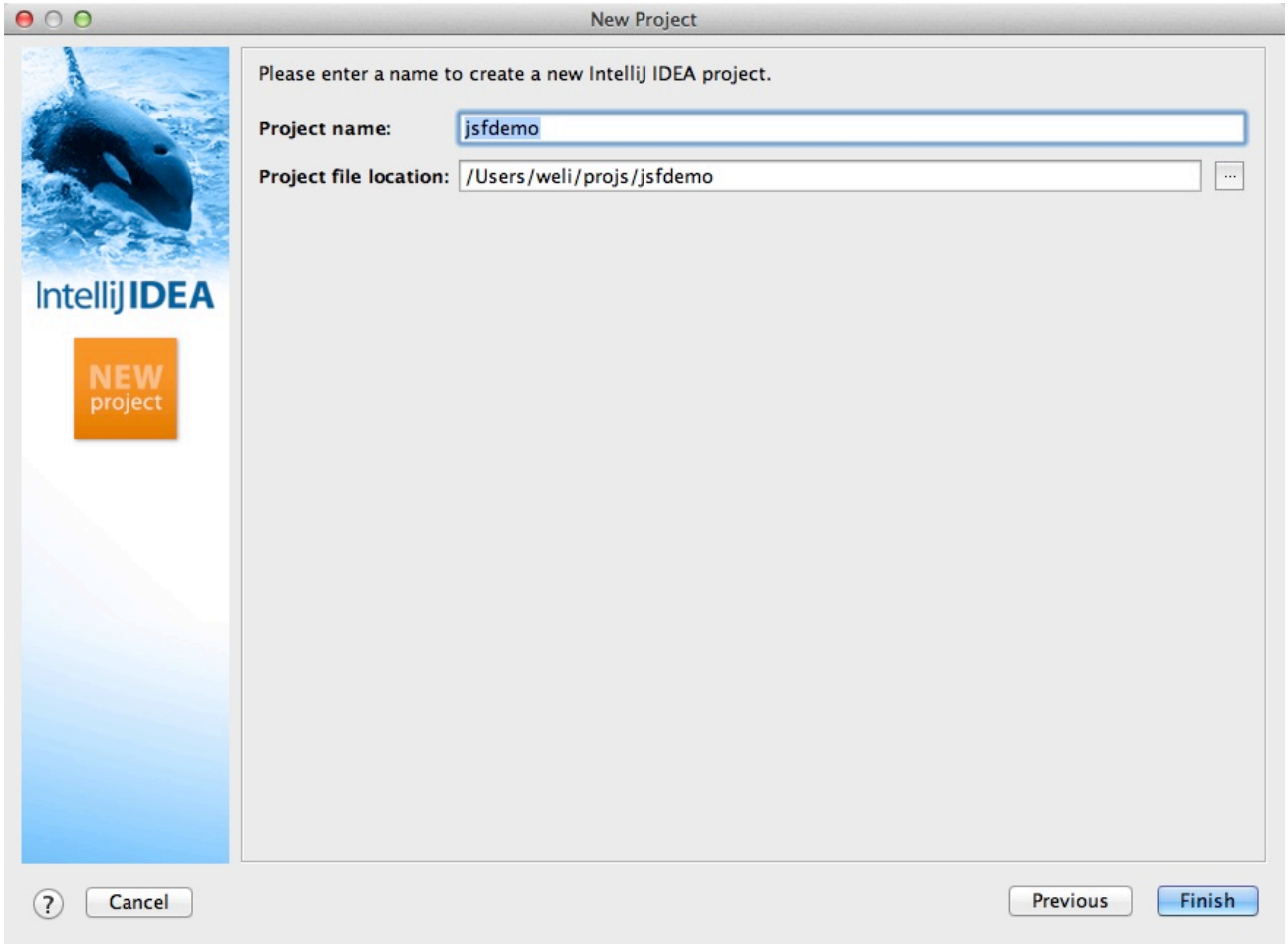
Then IntelliJ will ask you the position of the project you want to import. In 'Root directory' input your project's directory and leave other options as default:



For next step, just click 'Next':



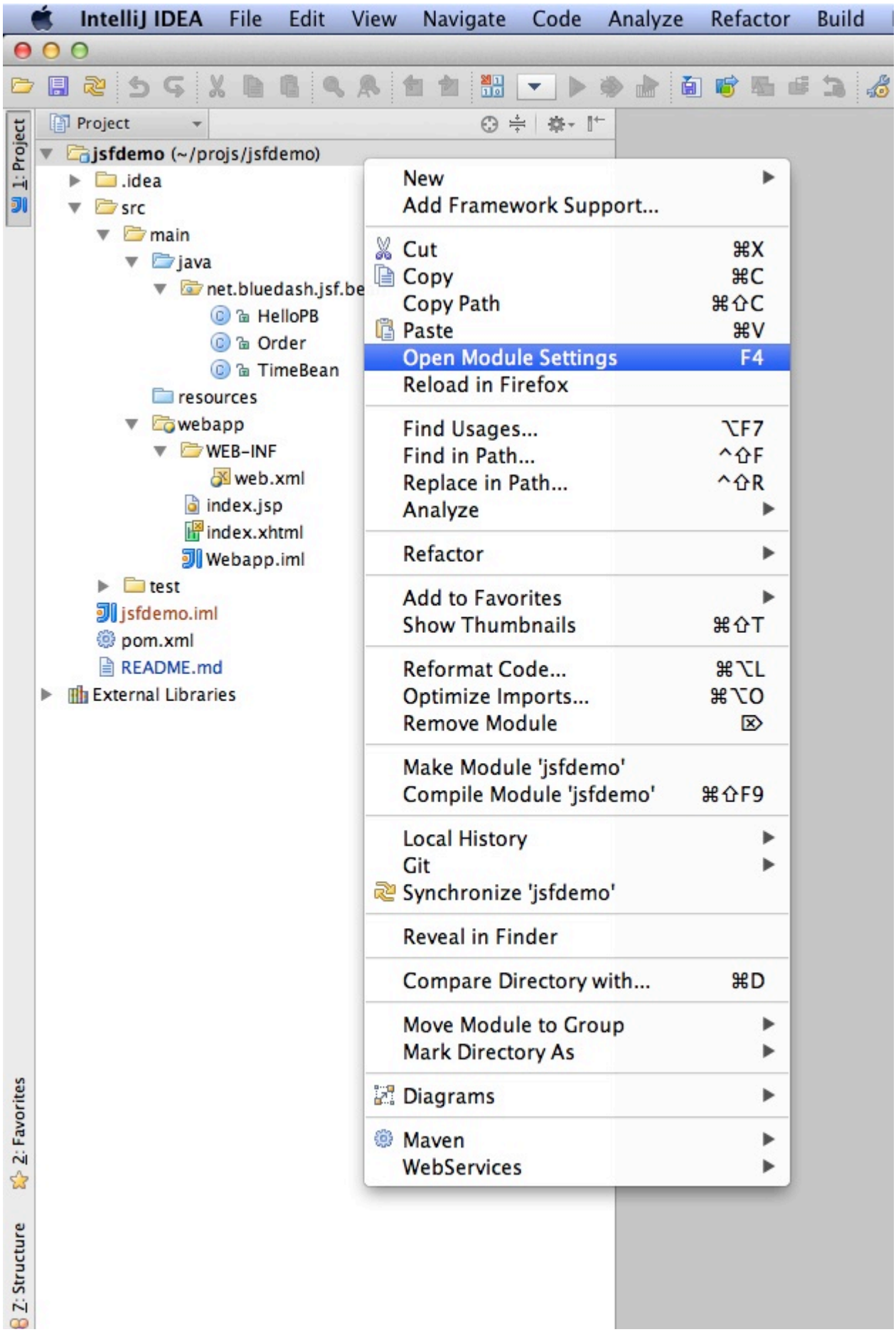
Finally click 'Finish':

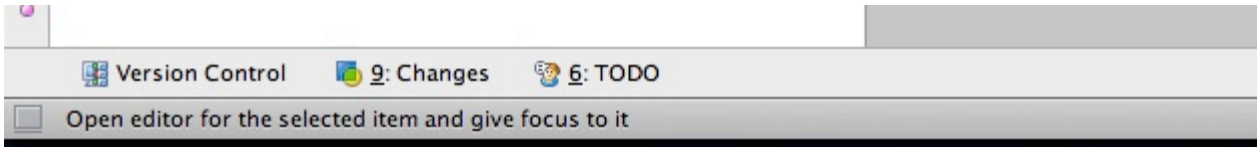


Hooray! We've imported the project into IntelliJ now

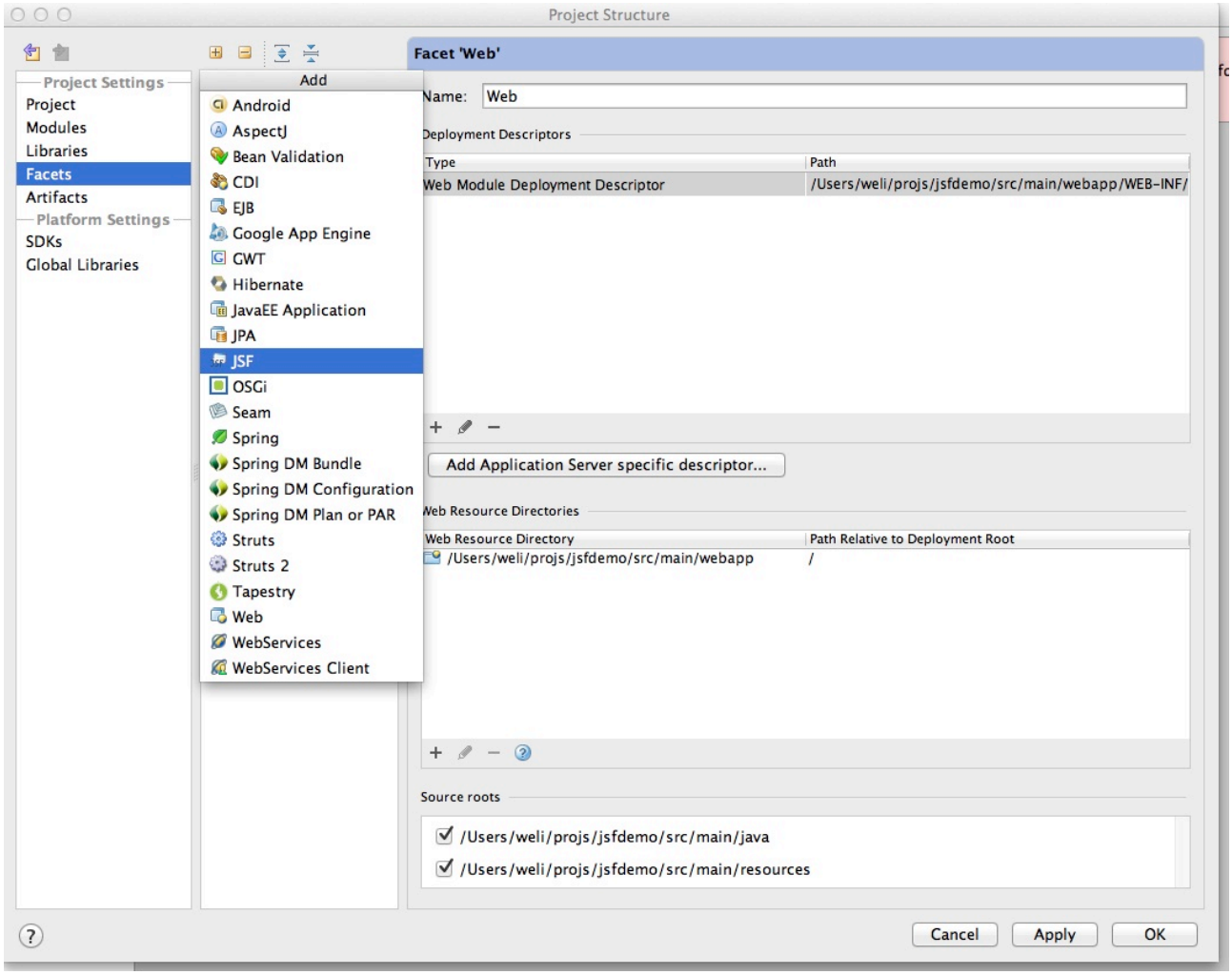
Adding IntelliJ JSF support to project

Let's see how to use IntelliJ and AS7 to debug the project. First we need to add 'JSF' facet into project. Open project setting:

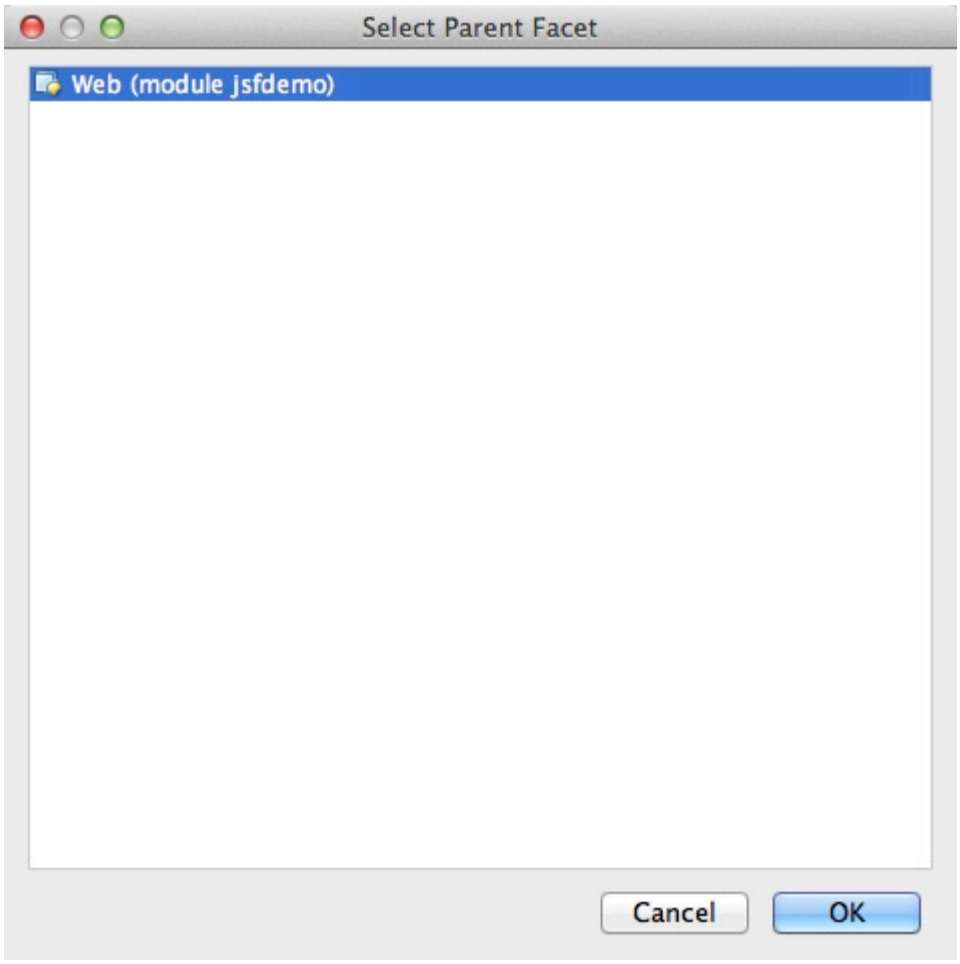

IntelliJ IDEA
File
Edit
View
Navigate
Code
Analyze
Refactor
Build
Project
jsfdemo (~/projs/jsfdemo)
.idea
src
main
java
net.bluedash.jsf.be
HelloPB
Order
TimeBean
resources
webapp
WEB-INF
web.xml
index.jsp
index.xhtml
Webapp.iml
test
jsfdemo.iml
pom.xml
README.md
External Libraries
1: Project
2: Favorites
Z: Structure
New
Add Framework Support...
Cut ⌘X
Copy ⌘C
Copy Path ⌘⇧C
Paste ⌘V
Open Module Settings F4
Reload in Firefox
Find Usages... ⌥F7
Find in Path... ^⇧F
Replace in Path... ^⇧R
Analyze
Refactor
Add to Favorites
Show Thumbnails ⌘⇧T
Reformat Code... ⌘⌥L
Optimize Imports... ⌘⌥O
Remove Module ⌦
Make Module 'jsfdemo'
Compile Module 'jsfdemo' ⌘⇧F9
Local History
Git
Synchronize 'jsfdemo'
Reveal in Finder
Compare Directory with... ⌘D
Move Module to Group
Mark Directory As
Diagrams
Maven
WebServices



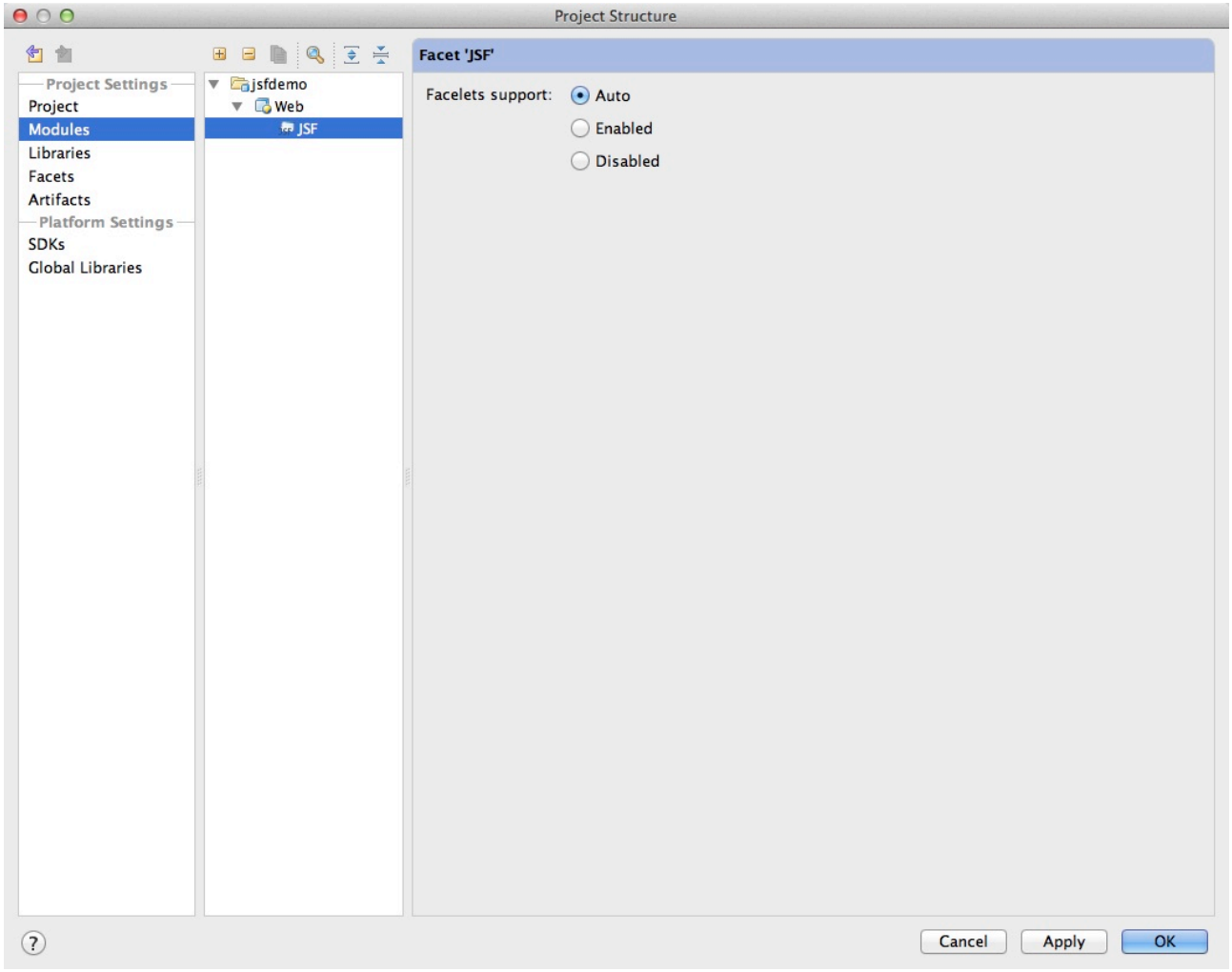
Click on 'Facets' section on left; Select 'Web' facet that we already have, and click the '+' on top, choose 'JSF':



Select 'Web' as parent facet:



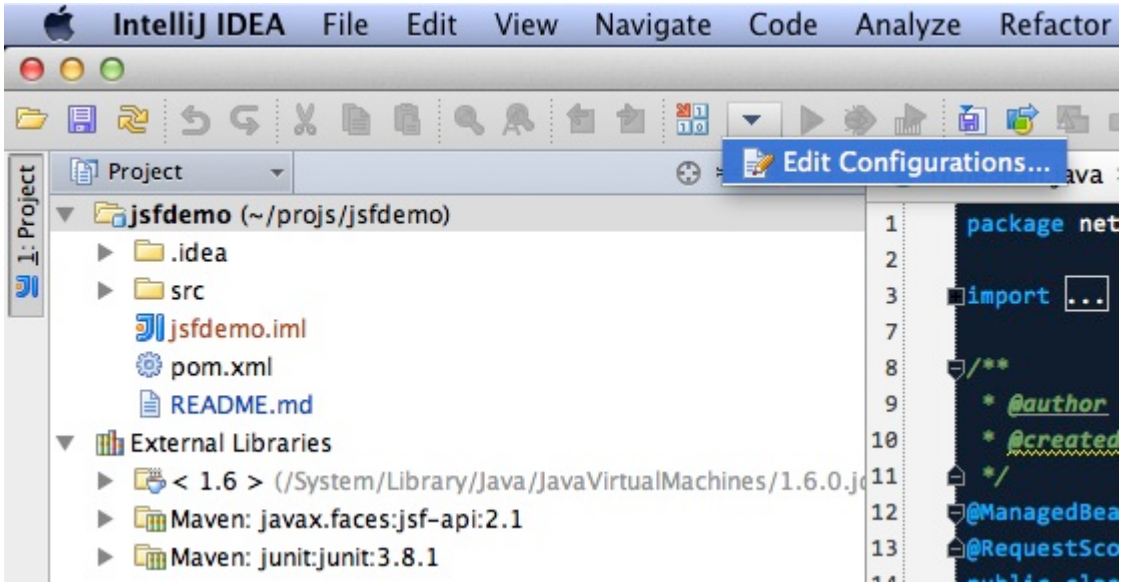
Click 'Ok':



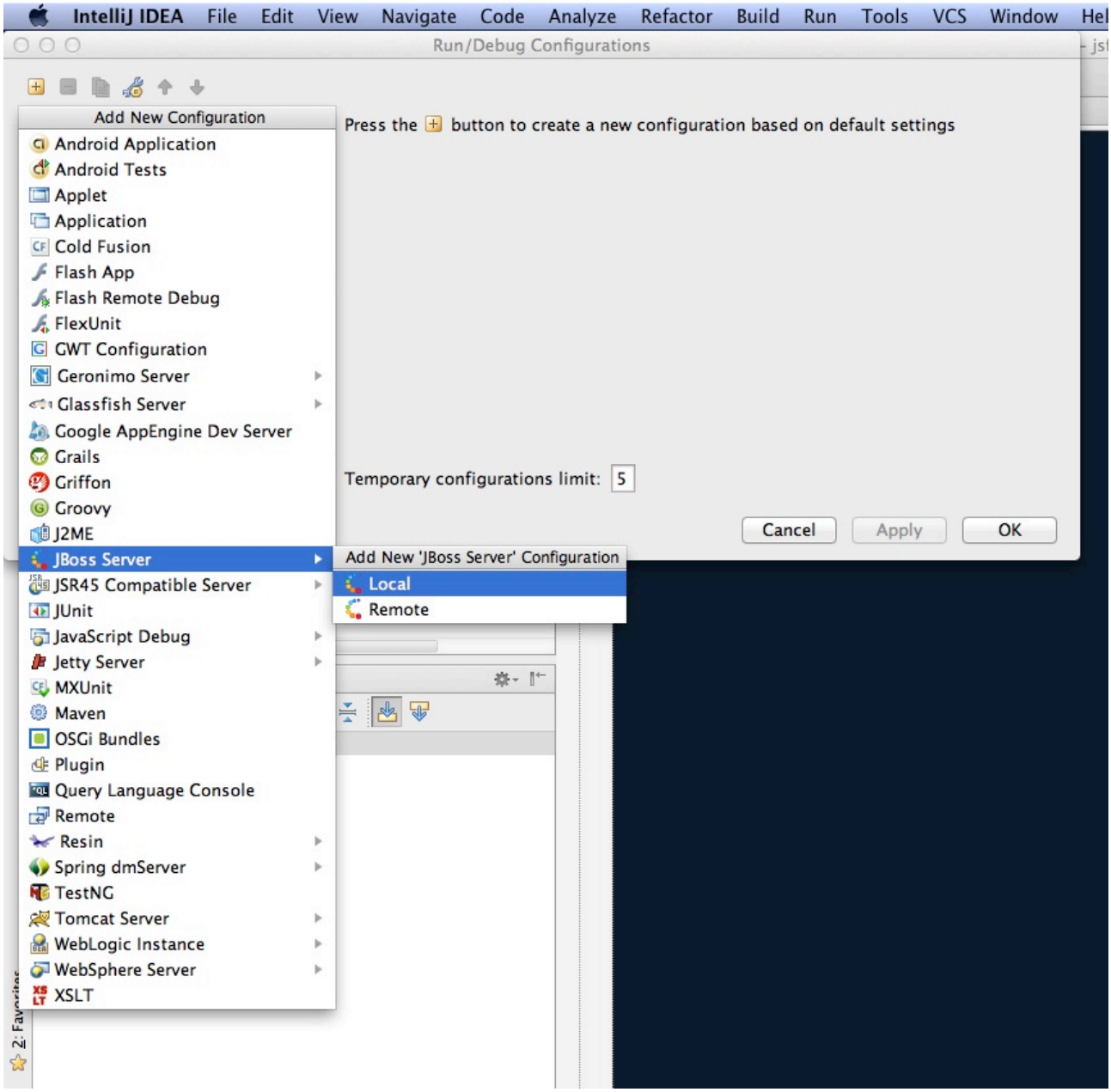
Now we have enabled IntelliJ's JSF support for project.

Add JBoss AS7 to IntelliJ

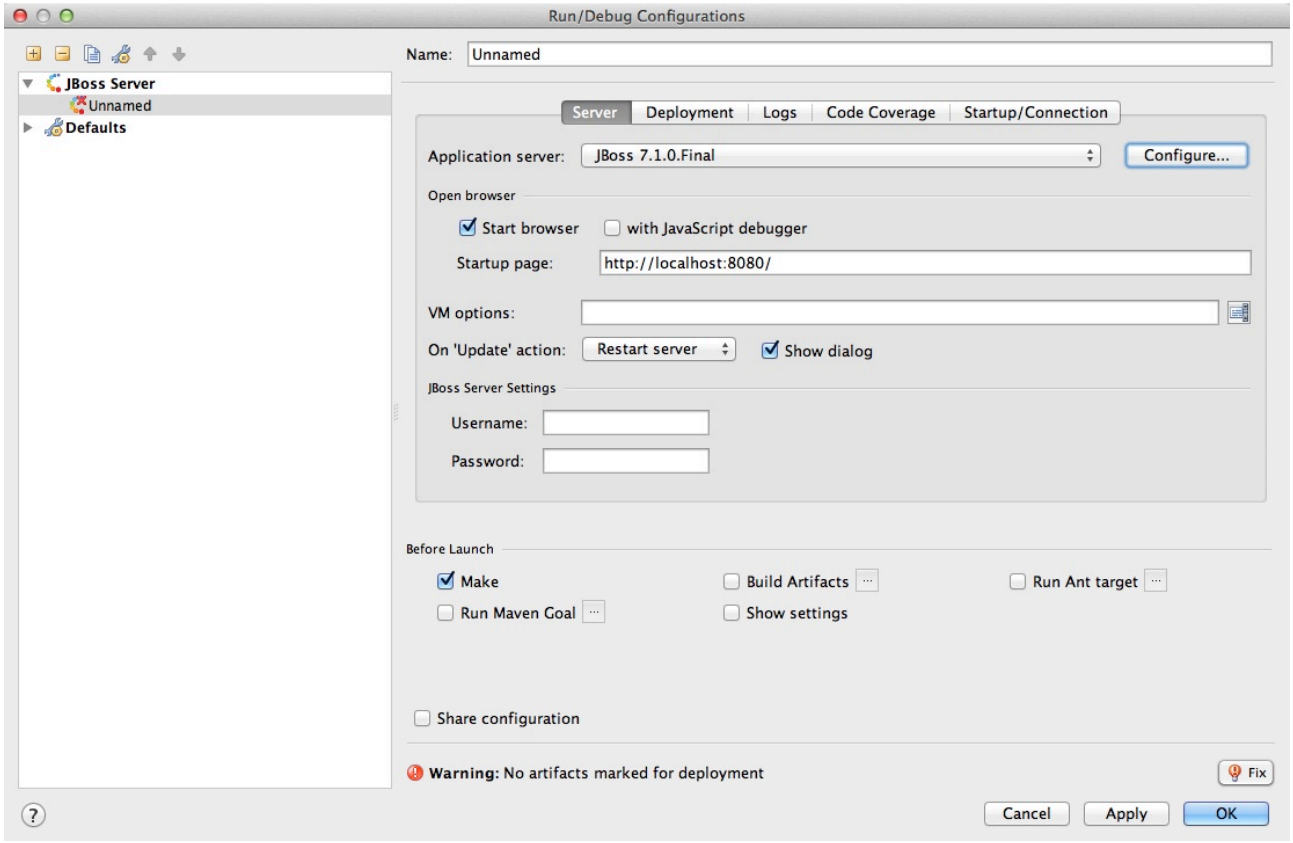
Let's add JBoss AS7 into IntelliJ and use it to debug our project. First please choose 'Edit Configuration' in menu tab:



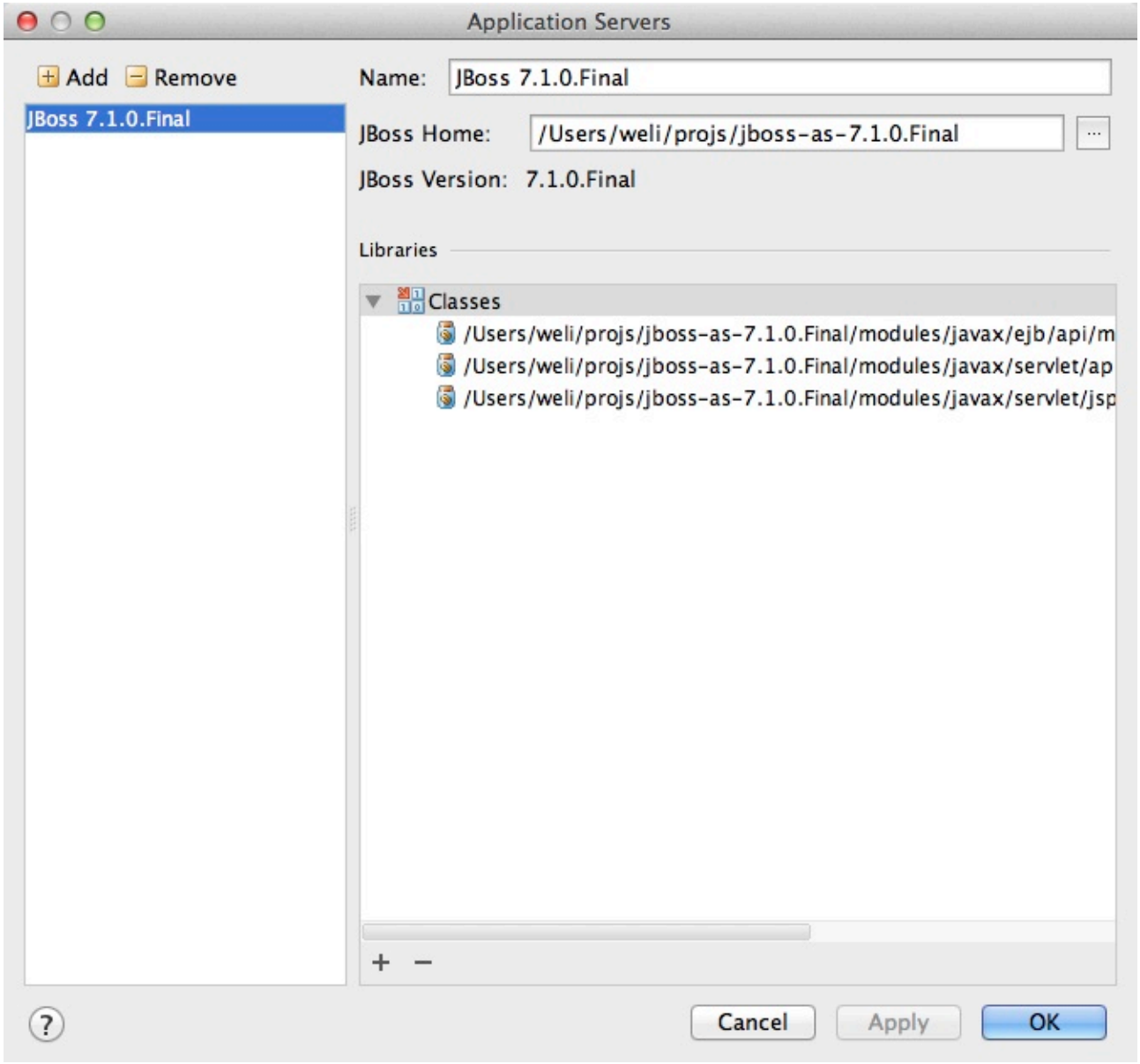
Click '+' and choose 'JBoss Server' -> 'Local':



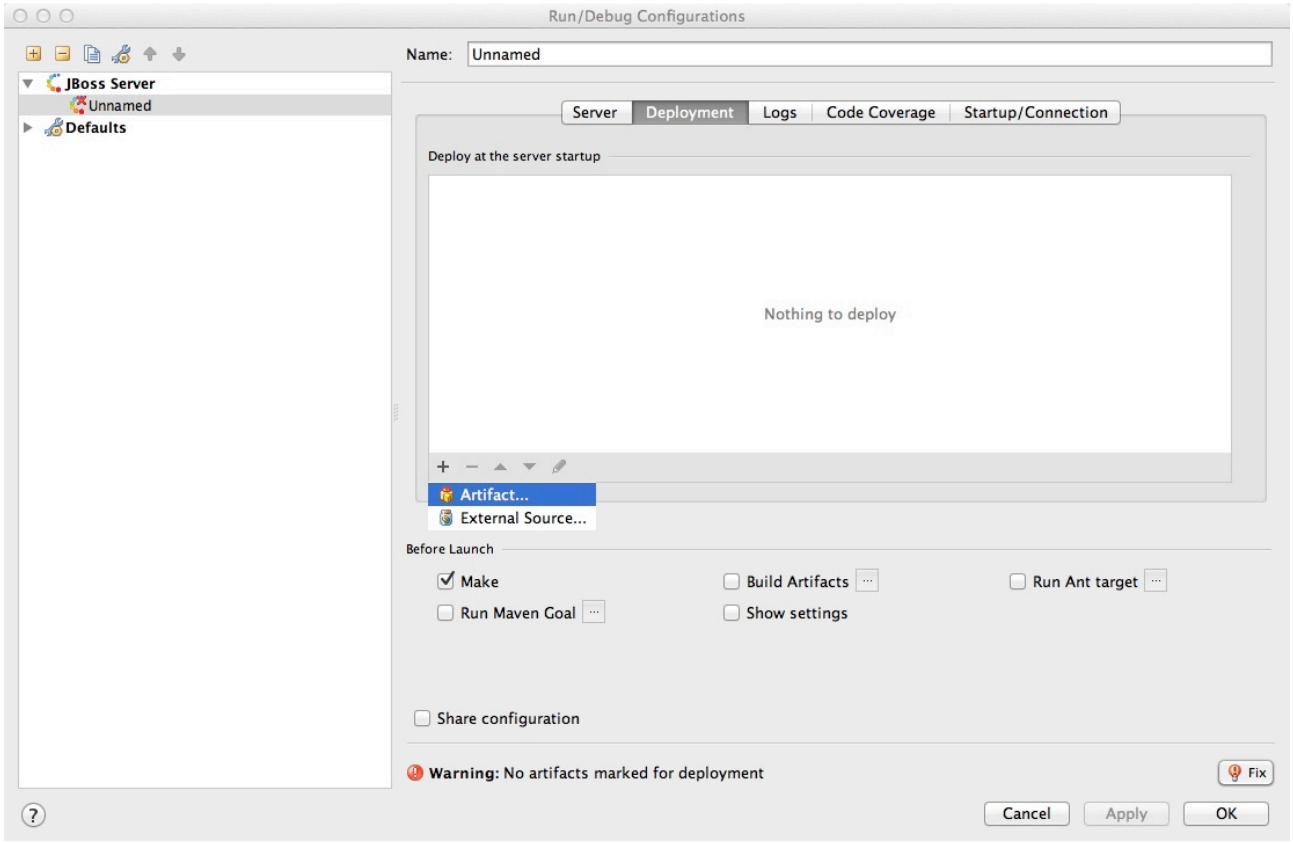
Click 'configure':



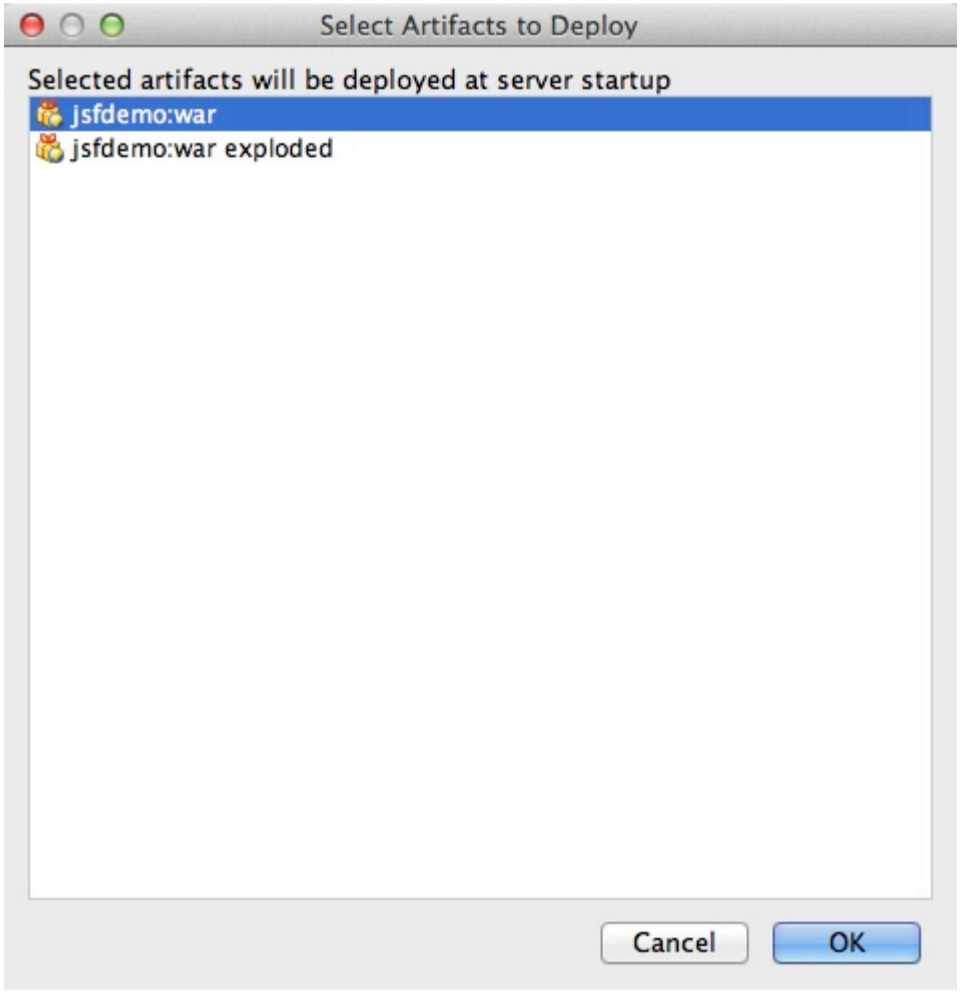
and choose your JBoss AS7:



Now we need to add our project into deployment. Click the 'Deployment' tab:



Choose 'Artifact', and add our project:



Leave everything as default and click 'Ok', now we've added JBoss AS7 into IntelliJ

Debugging project with IntelliJ and AS7

Now comes the fun part. To debug our project, we cannot directly use the 'debug' feature provided by IntelliJ right now(maybe in the future version this problem could be fixed). So now we should use the debugging config provided by AS7 itself to enable JPDA feature, and then use the remote debug function provided by IntelliJ to get things done. Let's dive into the details now:

First we need to enable JPDA config inside AS7, open 'bin/standalone.conf' and find following lines:

```
# Sample JPDA settings for remote socket debugging
#JAVA_OPTS="$JAVA_OPTS -Xrunjdwp:transport=dt_socket,address=8787,server=y,suspend=n"
```

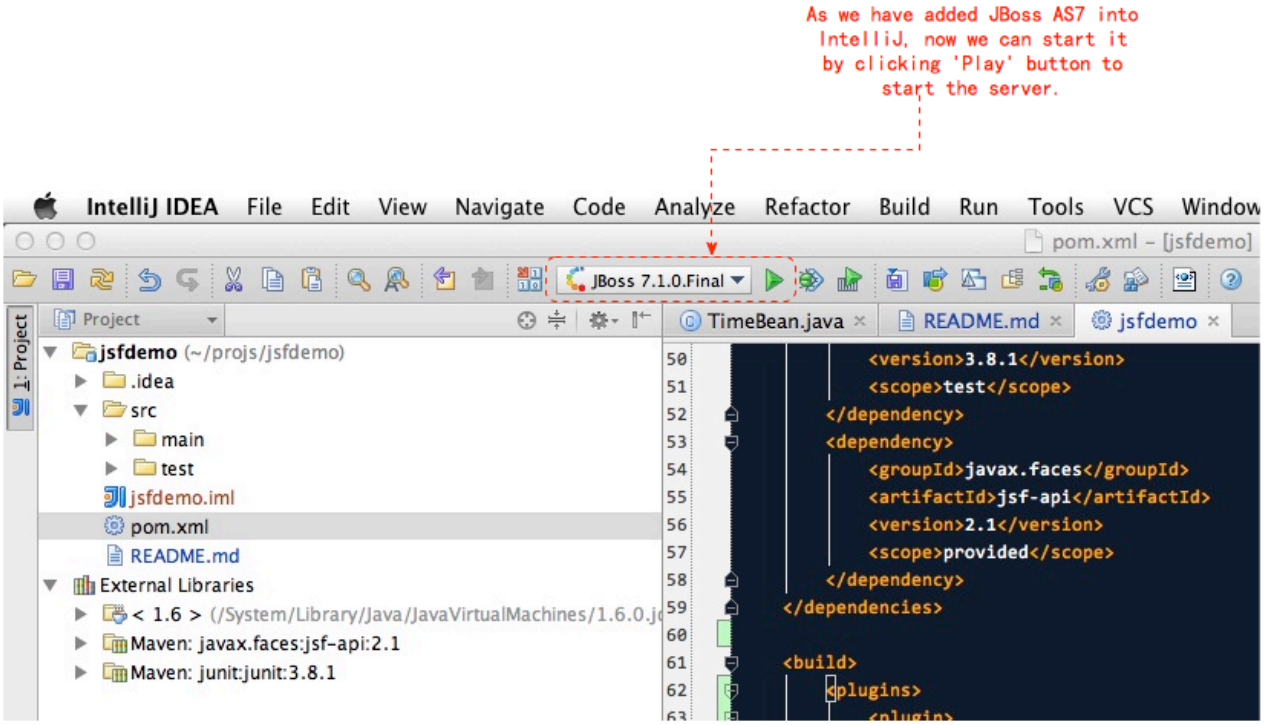
Enable the above config by removing the leading hash sign:

```
# Sample JPDA settings for remote socket debugging
JAVA_OPTS="$JAVA_OPTS -Xrunjdwp:transport=dt_socket,address=8787,server=y,suspend=n"
```

With WildFly you can directly start the server in debug mode:

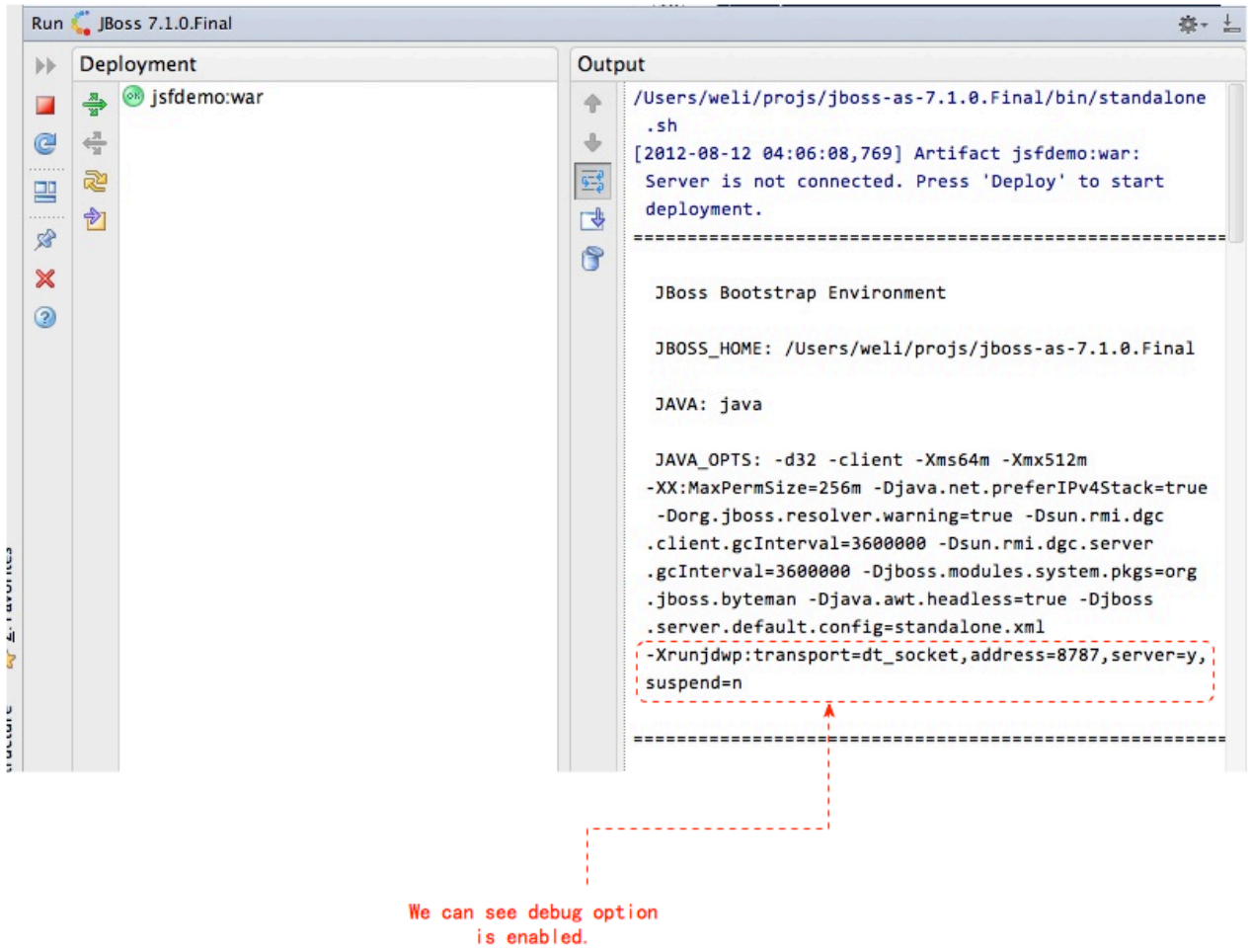
```
bin/standalone.sh --debug --server-config=standalone.xml
```

Now we start AS7 in IntelliJ:

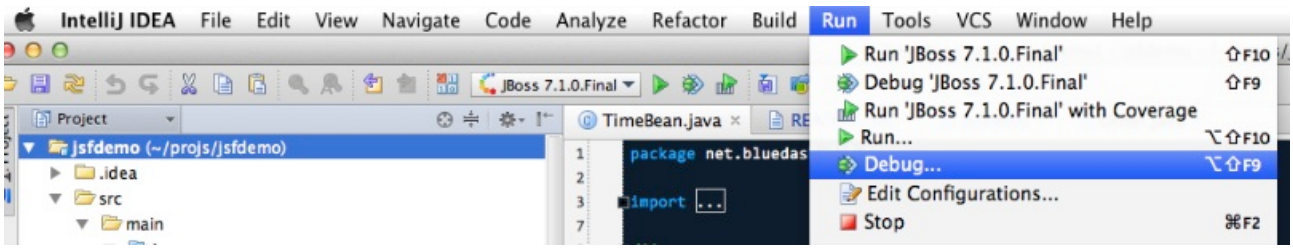


Please note we should undeploy the existing 'jsfdemo' project in AS7 as we've added by maven jboss deploy plugin before. Or AS7 will tell us there is already existing project with same name so IntelliJ could not deploy the project anymore.

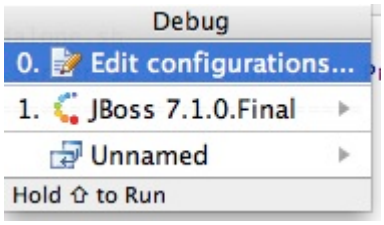
If the project start correctly we can see from the IntelliJ console window, and please check the debug option is enabled:



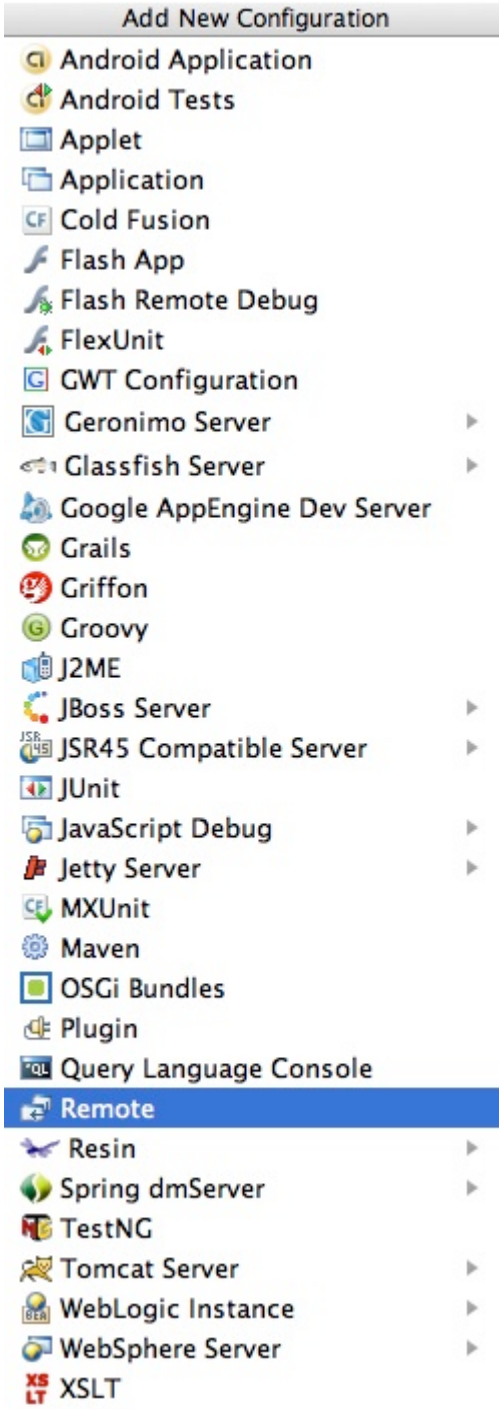
Now we will setup the debug configuration, click 'debug' option on menu:



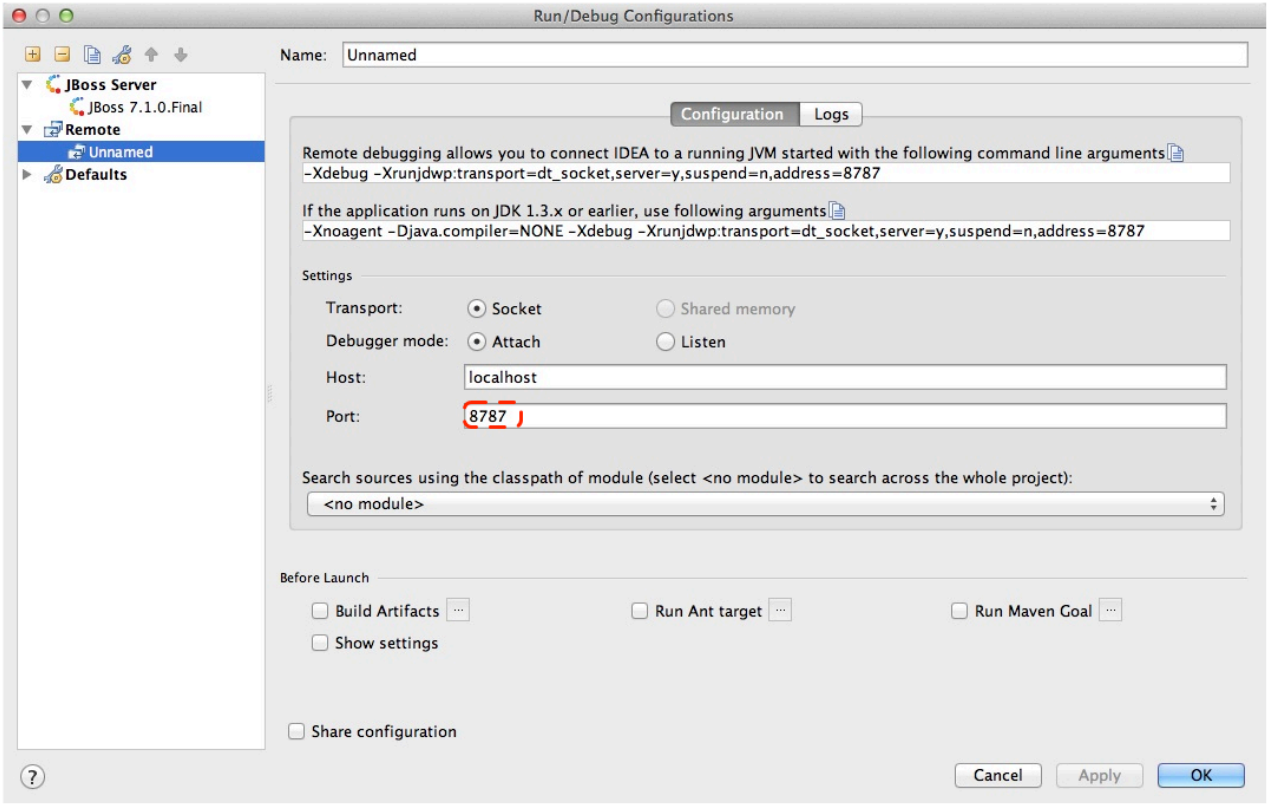
Choose 'Edit Configurations':



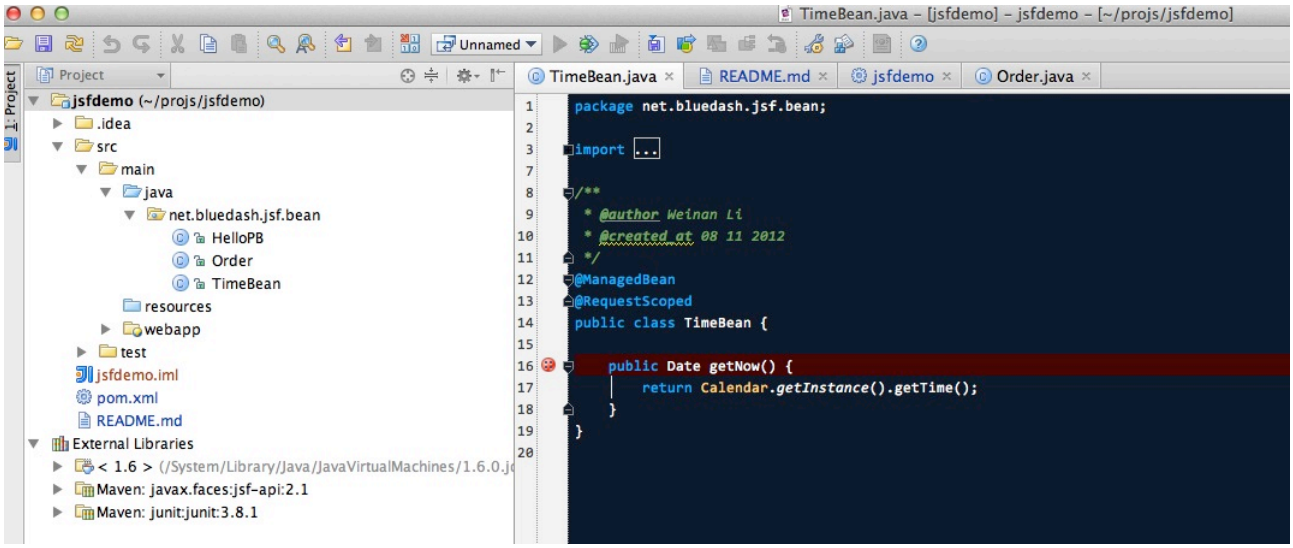
Then we click 'Add' and choose Remote:



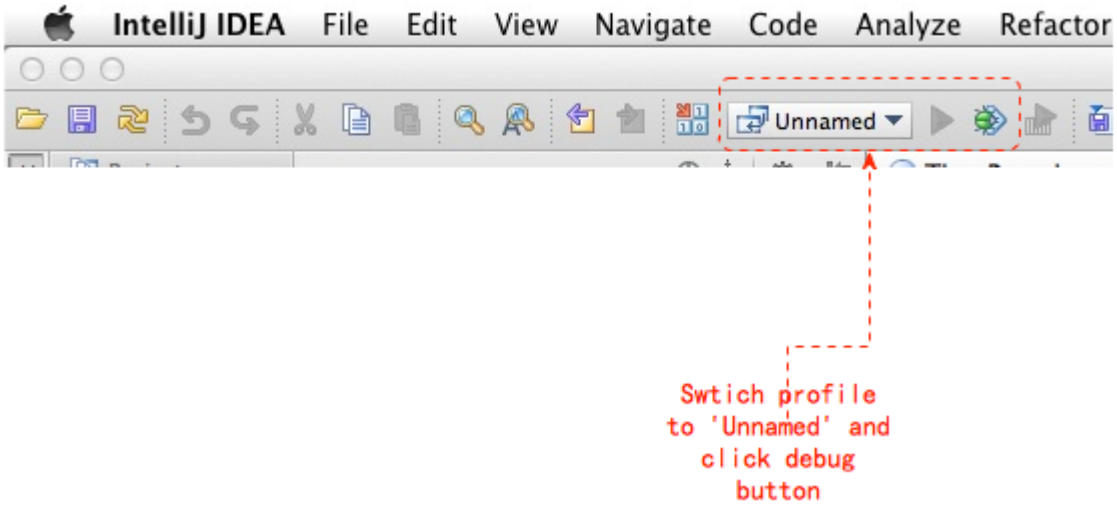
Set the 'port' to the one you used in AS7 config file 'standalone.conf':



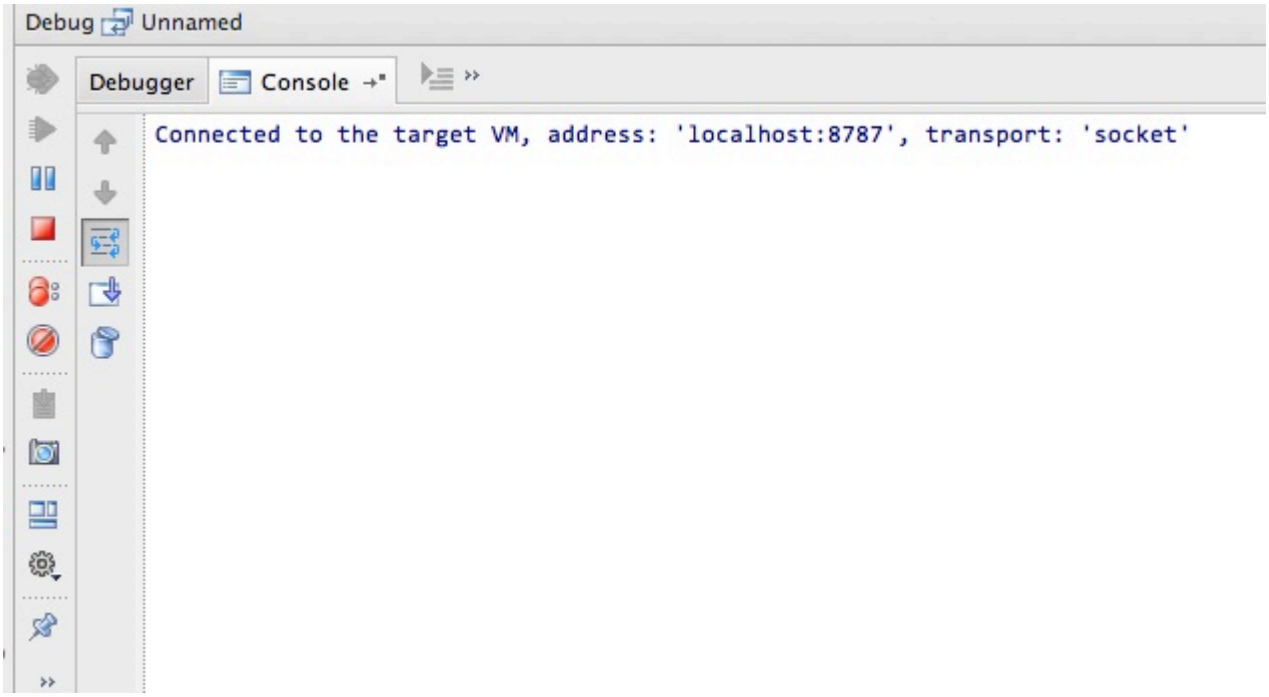
Leave other configurations as default and click 'Ok'. Now we need to set breakpoints in project, let's choose TimeBean.java and set a breakpoint on 'getNow()' method by clicking the left side of that line of code:



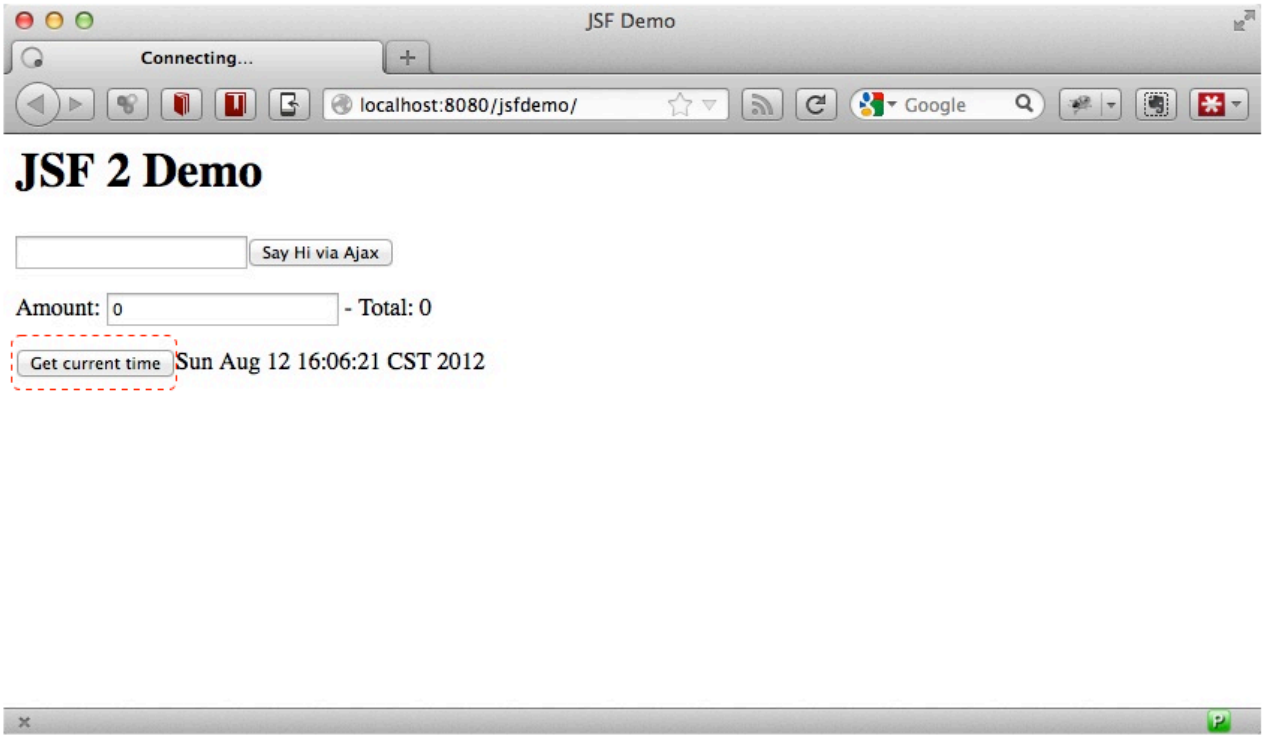
Now we can use the profile to do debug:



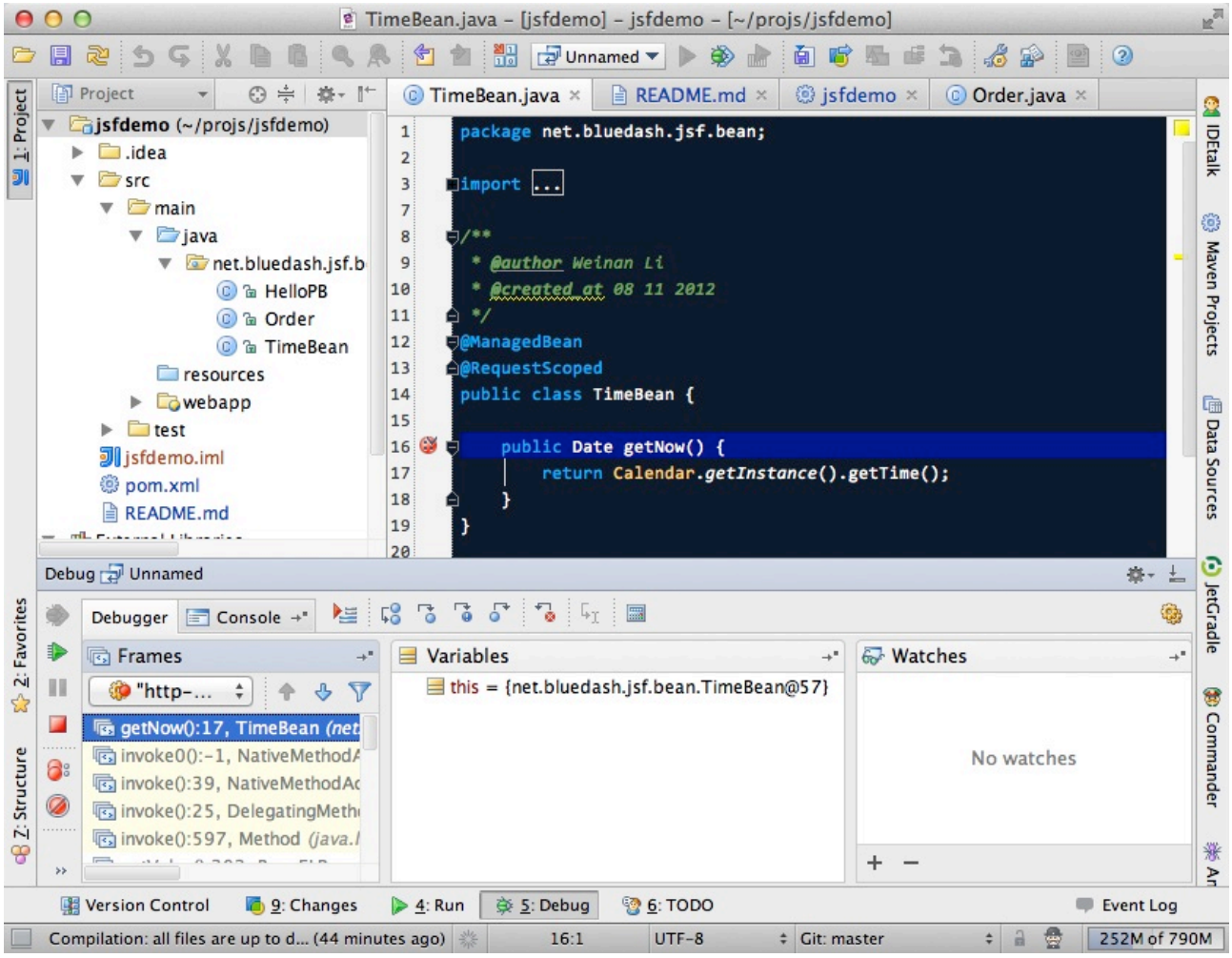
If everything goes fine we can see the console output:



Now we go to web browser and see our project's main page, try to click on 'Get current time':



Then IntelliJ will popup and the code is pausing on break point:



And we could inspect our project now.

Conclusion

In this article I've shown to you how to use maven to create a project using JSF and deploy it in JBoss AS7, and I've also talked about the usage of IntelliJ during project development phase. Hope the contents are practical and helpful to you

References

- *JBoss AS7: Using JPDA to debug the AS source code*
- *Importing JBoss 7 Bundled Dependency Versions Through Maven*
- *Maven Getting Started - Developers*
- *JSF 2.1 project using Eclipse and Maven 2:http*
- *Practical RichFaces*
- *Oracle Mojarra JavaServer Faces*
- *JBoss AS7 Maven Plugin*

8.8.2 Getting Started Developing Applications Presentation & Demo

Introduction

This document is a “script” for use with the quickstarts associated with the Getting Started Developing Applications Guide. It can be used as the basis for demoing/explaining the Java EE 6 programming model with JBoss AS 7.

There is an associated presentation – JBoss AS - Getting Started Developing Applications – which can be used to introduce the Java EE 6 ecosystem.

The emphasis here is on the programming model, not on OAM/dev-ops, performance etc.

Prerequisites for using the script

- JBoss AS 7 downloaded and installed
- Eclipse Indigo with m2eclipse and JBoss Tools installed
- The quickstarts downloaded and imported into Eclipse
- Make sure `$JBOSS_HOME` is set.
- Make sure `src/test/resources/arquillian.xml` has the correct path to your JBoss AS install for kitchensink
- Make sure your font size is set in Eclipse so everyone can read the text!

Import examples into Eclipse and set up JBoss AS

TODO

The Helloworld Quickstart

Introduction

This quickstart is extremely basic, and is really useful for nothing more than showing than the app server is working properly, and our deployment mechanism is working. We recommend you use this quickstart to demonstrate the various ways you can deploy apps to JBoss AS 7.

Using Maven

1. Start JBoss AS 7 from the console

```
$JBOSS_HOME/bin/standalone.sh
```

2. Deploy the app using Maven

```
mvn clean package jboss-as:deploy
```

The quickstarts use the jboss-as maven plugin to deploy and undeploy applications. This plugin uses the JBoss AS Native Java Detyped Management API to communicate with the server. The Detyped API is used by management tools to control an entire domain of servers, and exposes only a small number of types, allowing for backwards and forwards compatibility.

3. Show the app has deployed in the terminal
4. Visit http://localhost:8080/jboss-as-helloworld
5. Undeploy the app using Maven

```
mvn jboss-as:undeploy
```

Using the Command Line Interface (CLI)

1. Start JBoss AS 7 from the console (if not already running)

```
$JBOSS_HOME/bin/standalone.sh
```

2. Build the war

```
mvn clean package
```

3. Start the CLI

```
$JBOSS_HOME/bin/jboss-admin.sh --connect
```

The command line also uses the Deptyped Management API to communicate with the server. It's designed to be as "unixy" as possible, allowing you to "cd" into nodes, with full tab completion etc. The CLI allows you to deploy and undeploy applications, create JMS queues, topics etc., create datasources (normal and XA). It also fully supports the domain node.

4. Deploy the app

```
deploy target/jboss-as-helloworld.war
```

5. Show the app has deployed

```
undeploy jboss-as-helloworld.war
```

Using the web management interface

1. Start JBoss AS 7 from the console (if not already running)

```
$JBOSS_HOME/bin/standalone.sh
```

2. Build the war

```
mvn clean package
```

3. Open up the web management interface http://localhost:9990/console

> The web maangement interface offers the same functionality as the CLI (and again uses the Detyped Management API), but does so using a pretty GWT interface! You can set up virtual servers, interrogate sub systems and more.

4. Navigate `Manage Deployments -> Add content`. Click on choose file and locate `helloworld/target/jboss-as-helloworld.war`.
5. Click `Next` and `Finish` to upload the war to the server.
6. Now click `Enable` and `Ok` to start the application
7. Switch to the console to show it deployed
8. Now click `Remove`

Using the filesystem

1. Start JBoss AS 7 from the console (if not already running)

```
$JBOSS_HOME/bin/standalone.sh
```

2. Build the war

```
mvn clean package
```

> Of course, you can still use the good ol' file system to deploy. Just copy the file to `$JBOSS_HOME/standalone/deployments`.

3. Copy the war

```
cp target/jboss-as-helloworld.war $JBOSS_HOME/standalone/deployments
```

4. Show the war deployed

> The filesystem deployment uses marker files to indicate the status of a deployment. As this deployment succeeded we get a `$JBOSS_HOME/standalone/deployments/jboss-as-helloworld.war.deployed` file. If the deployment failed, you would get a `.failed` file etc.

5. Undeploy the war

```
rm $JBOSS_HOME/standalone/deployments/jboss-as-helloworld.war.deployed
```

6. Show the deployment stopping!
7. Start and stop the appserver, show that the deployment really is gone!

> This gives you much more precise control over deployments than before

Using Eclipse

1. Add a JBoss AS server
    1. Bring up the Server view
    2. Right click in it, and choose `New -> Server`
    3. Choose JBoss AS 7.0 and hit Next
    4. Locate the server on your disc
    5. Hit Finish
2. Start JBoss AS in Eclipse
    1. Select the server
    2. Click the Run button
3. Deploy the app
    1. right click on the app, choose `Run As -> Run On Server`
    2. Select the AS 7 instance you want to use
    3. Hit finish
4. Load the app at http://localhost:8080/jboss-as-helloworld

Digging into the app

1. Open up the helloworld quickstart in Eclipse, and open up `src/main/webapp`.
2. Point out that we don't require a `web.xml` anymore!
3. Show `beans.xml` and explain it's a marker file used to JBoss AS to enable CDI (open it, show that it is empty)
4. Show `index.html`, and explain it is just used to kick the user into the app (open it, show the meta-refresh)
5. Open up the `pom.xm` - and emphasise that it's pretty simple.
    1. There is no parent pom, everything for the build is **here**
    2. Show that we are enabling the JBoss Maven repo - explain you can do this in your POM or in system wide (`settings.xml`)
    3. Show the `dependencyManagement` section. Here we import the JBoss AS 7 Web Profile API. Explain that this gives you all the versions for all of the JBoss AS 7 APIs that are in the web profile. Explain we could also depend on this directly, which would give us the whole set of APIs, but that here we've decided to go for slightly tighter control and specify each dependency ourselves
    4. Show the import for CDI, JSR-250 and Servlet API. Show that these are all provided - we are depending on build in server implementations, not packaging this stuff!
    5. Show the plugin sections - nothing that exciting here, the war plugin is out of date and requires you to provide `web.xml`  , configure the JBoss AS Maven Plugin, set the Java version to 6.
6. Open up `src/main/java` and open up the `HelloWorldServlet`.
    1. Point out the `@WebServlet` - explain this one annotation removes about 8 lines of XML - no need to separately map a path either. This is much more refactor safe
    2. Show that we can inject services into a Servlet
    3. Show that we use the service (line 41)
       #Cmd-click on `HelloService`
    4. This is a CDI bean - very simple, no annotations required!
    5. Explain injection
        1. Probably used to string based bean resolution
        2. This is typesafe (refactor safe, take advantage of the compiler and the IDE - we just saw that!)
        3. When CDI needs to inject something, the first thing it looks at is the type - and if the type of the injection point is assignable from a bean, CDI will inject that bean

The numberguess quickstart

Introduction

This quickstart adds in a "complete" view layer into the mix. Java EE ships with a JSF. JSF is a server side rendering, component orientated framework, where you write markup using an HTML like language, adding in dynamic behavior by binding components to beans in the back end. The quickstart also makes more use of CDI to wire the application together.

Run the app

1. Start JBoss AS in Eclipse
2. Deploy it using Eclipse - just right click on the app, choose `Run As -> Run On Server`
3. Select the AS 7 instance you want to use
4. Hit finish
5. Load the app at http://localhost:8080/jboss-as-numberguess
6. Make a few guesses

Deployment descriptors src/main/webapp/WEB-INF

Emphasize the lack of them!

No need to open any of them, just point them out

1. `web.xml` - don't need it!
2. `beans.xml` - as before, marker file
3. `faces-config.xml` - nice feature from AS7 - we can just put `faces-config.xml` into the WEB-INF and it enables JSF (inspiration from CDI)
4. `pom.xml` we saw this before, this time it's the same but adds in JSF API

Views

1. `index.html` - same as before, just kicks us into the app
2. `home.xhtml`
    1. Lines 19 - 25 – these are messages output depending on state of beans (minimise coupling between controller and view layer by interrogating state, not pushing)
3. Line 20 – output any messages pushed out by the controller
4. Line 39 - 42 – the input field is bound to the guess field on the game bean. We validate the input by calling a method on the game bean.
5. Line 43 - 45 – the command button is used to submit the form, and calls a method on the game bean
6. Line 48, 49, The reset button again calls a method on the game bean

Beans

1. `Game.java` – this is the main controller for the game. App has no persistence etc.
    1. `@Named` – As we discussed CDI is typesafe, (beans are injected by type) but sometimes need to access in a non-typesafe fashion. @Named exposes the Bean in EL - and allows us to access it from JSF
    2. `@SessionScoped` – really simple app, we keep the game data in the session - to play two concurrent games, need two sessions. This is not a limitation of CDI, but simply keeps this demo very simple. CDI will create a bean instance the first time the game bean is accessed, and then always load that for you
    3. `@Inject maxNumber` – here we inject the maximum number we can guess. This allows us to externalize the config of the game
    4. `@Inject rnadomNumber` – here we inject the random number we need to guess. Two things to discuss here
        1. Instance - normally we can inject the object itself, but sometimes it's useful to inject a "provider" of the object (in this case so that we can get a new random number when the game is reset!). Instance allows us to `get()` a new instance when needed
        2. Qualifiers - now we have two types of Integer (CDI auto-boxes types when doing injection) so we need to disambiguate. Explain qualifiers and development time approach to disambiguation. You will want to open up `@MaxNumber` and `@Random` here.
    5. `@PostConstruct` – here is our reset method - we also call it on startup to set up initial values. Show use of `Instance.get()`.
2. `Generator.java` This bean acts as our random number generator.
3. `@ApplicationScoped` explain about other scopes available in CDI + extensibility.
    1. `next()` Explain about producers being useful for determining bean instance at runtime
    2. `getMaxNumber()` Explain about producers allowing for loose coupling

The login quickstart

Introduction

The login quickstart builds on the knowledge of CDI and JSF we have got from numberguess. New stuff we will learn about is how to use JPA to store data in a database, how to use JTA to control transactions, and how to use EJB for declarative TX control.

Run the app

1. Start JBoss AS in Eclipse
2. Deploy it using Eclipse - just right click on the app, choose `Run As -> Run On Server`
3. Select the AS 7 instance you want to use
4. Hit finish
5. Load the app at http://localhost:8080/jboss-as-login
6. Login as admin/admin
7. Create a new user

Deployment Descriptors

1. Show that we have the same ones we are used in `src/main/webapp` – `beans.xml`, `faces-config.xml`
2. We have a couple of new ones in `src/main/resources`
    1. `persistence.xml`. Not too exciting. We are using a datasource that AS7 ships with. It's backed by the H2 database and is purely a sample datasource to use in sample applications. We also tell Hibernate to auto-create tables - as you always have.
    2. `import.sql` Again, the same old thing you are used to in Hibernate - auto-import data when the app starts.
3. `pom.xml` is the same again, but just adds in dependencies for JPA, JTA and EJB

Views

1. `template.xhtml` One of the updates added to JSF 2.0 was templating ability. We take advantage of that in this app, as we have multiple views
    1. Actually nothing too major here, we define the app "title" and we could easily define a common footer etc. (we can see this done in the kitchensink app)
    2. The `ui:insert` command inserts the actual content from the templated page. #`home.xhtml`
    3. Uses the template
    4. Has some input fields for the login form, button to login and logout, link to add users.
    5. Binds fields to credentials bean}}
    6. Buttons link to login bean which is the controller
2. `users.xhtml`
    1. Uses the template
    2. Displays all users using a table
    3. Has a form with input fields to add users.
    4. Binds fields to the newUser bean
    5. Methods call on userManager bean

Beans

1. `Credentials.java` Backing bean for the login form field, pretty trivial. It's request scoped (natural for a login field) and named so we can get it from JSF.
2. `Login.java`
    1. Is session scoped (a user is logged in for the length of their session or until they log out}}
    2. Is accessible from EL
    3. Injects the current credentials
    4. Uses the userManager service to load the user, and sends any messages to JSF as needed
    5. Uses a producer method to expose the @LoggedIn user (producer methods used as we don't know which user at development time)
3. `User.java` Is a pretty straightforward JPA entity. Mapped with `@Entity`, has an natural id.
4. `UserManager.java` This is an interface, and by default we use the ManagedBean version, which requires manual TX control
5. `ManagedBeanUserManager.java` - accessible from EL, request scoped.
    1. Injects a logger (we'll see how that is produced in a minute)
    2. Injects the entity manager (again, just a min)
    3. Inject the UserTransaction (this is provided by CDI)
    4. `getUsers()` standard JPA-QL that we know and love - but lots of ugly TX handling code.
    5. Same for `addUser()` and `findUser()` methods - very simple JPA but...
    6. Got a couple of producer methods.
        1. `getUsers()` is obvious - loads all the users in the database. No ambiguity - CDI takes into account generic types when injecting. Also note that CDI names respect JavaBean naming conventions
        2. `getNewUser()` is used to bind the new user form to from the view layer - very nice as it decreases coupling - we could completely change the wiring on the server side (different approach to creating the newUser bean) and no need to change the view layer.
6. `EJBUserManager.java`
    1. It's an alternative – explain alternatives, and that they allow selection of beans at deployment time
    2. Much simple now we have declarative TX control.
    3. Start to see how we can introduce EJB to get useful enterprise services such as declarative TX control
7. `Resources.java`
    1. {EntityManager}} - explain resource producer pattern

The kitchensink quickstart

Introduction

The kitchensink quickstart is generated from an archetype available for JBoss AS (tell people to check the [Getting Started Developing Applications] Guide for details). It demonstrates CDI, JSF, EJB, JPA (which we've seen before) and JAX-RS and Bean Validation as well. We add in Arquillian for testing.

Run the app

1. Start JBoss AS in Eclipse
2. Deploy it using Eclipse - just right click on the app, choose `Run As -> Run On Server`
3. Select the AS 7 instance you want to use
4. Hit finish
5. Load the app at http://localhost:8080/jboss-as-kitchensink
6. Register a member - make sure to enter an invalid email and phone - show bean validation at work
7. Click on the member URL and show the output from JAX-RS

Bean Validation

1. Explain the benefits of bean validation - need your data always valid (protect your data) AND good errors for your user. BV allows you to express once, apply often.
2. `index.xhtml`
    1. Show the input fields – no validators attached
    2. Show the message output
3. `Member.java`
    1. Hightlight the various validation annotations
4. Java EE automatically applies the validators in both the persistence layer and in your views

RS

1. `index.xhtml` - Show that URL generation is just manual
2. `JaxRsActivator.java` - simply activates JAX-RS
3. `Member.java` - add JAXB annotation to make JAXB process the class properly
4. `MemberResourceRESTService.java`
    1. `@Path` sets the JAX-RS resource
    2. JAX-RS services can use injection
    3. `@GET` methods are auto transformed to XML using JAXB
5. And that is it!

Arquillian

1. Make sure JBoss AS is running
2. `mvn clean test -Parq-jbossas-remote`
    1. Explain the difference between managed and remote
3. Make sure JBoss AS is stopped
4. `mvn clean test -Parq-jbossas-managed`
5. Start JBoss AS in Eclipse
6. Update the project to use the `arq-jbossas-remote` profile
7. Run the test from Eclipse
    1. Right click on test, `Run As -> JUnit Test`
8. `MemberRegistrationTest.java`
    1. Discuss micro deployments
    2. Explain Arquilian allows you to use injection
    3. Explain that Arquillian allows you to concentrate just on your test logic

9 Getting Started Guide

9.1 Getting Started with WildFly 10

WildFly 10 is the latest release in a series of JBoss open-source application server offerings. WildFly 10 is an exceptionally fast, lightweight and powerful implementation of the Java Enterprise Edition 7 Platform specifications. The state-of-the-art architecture built on the Modular Service Container enables services on-demand when your application requires them. The table below lists the Java Enterprise Edition 7 technologies and the technologies available in WildFly 10 server configuration profiles.

Java EE 7 Platform Technology	Java EE 7 Full Profile	Java EE 7 Web Profile	WildFly 10 Full Profile	WildFly 10 Web Profile
JSR-356: Java API for Web Socket	X	X	X	X
JSR-353: Java API for JSON Processing	X	X	X	X
JSR-340: Java Servlet 3.1	X	X	X	X
JSR-344: JavaServer Faces 2.2	X	X	X	X
JSR-341: Expression Language 3.0	X	X	X	X
JSR-245: JavaServer Pages 2.3	X	X	X	X
JSR-52: Standard Tag Library for JavaServer Pages (JSTL) 1.2	X	X	X	X
JSR-352: Batch Applications for the Java Platform 1.0	X	--	X	--
JSR-236: Concurrency Utilities for Java EE 1.0	X	X	X	X
JSR-346: Contexts and Dependency Injection for Java 1.1	X	X	X	X

JSR-330: Dependency Injection for Java 1.0	X	X	X	X
JSR-349: Bean Validation 1.1	X	X	X	X
JSR-345: Enterprise JavaBeans 3.2	X CMP 2.0 Optional	X (Lite)	X CMP 2.0 Not Available	X (Lite)
JSR-318: Interceptors 1.2	X	X	X	X
JSR-322: Java EE Connector Architecture 1.7	X	--	X	X
JSR-338: Java Persistence 2.1	X	X	X	X
JSR-250: Common Annotations for the Java Platform 1.2	X	X	X	X
JSR-343: Java Message Service API 2.0	X	--	X	--
JSR-907: Java Transaction API 1.2	X	X	X	X
JSR-919: JavaMail 1.5	X	--	X	X
JSR-339: Java API for RESTFul Web Services 2.0	X	X	X	X
JSR-109: Implementing Enterprise Web Services 1.3	X	--	X	--
JSR-224: Java API for XML-Based Web Services 2.2	X	X	X	X
JSR-181: Web Services Metadata for the Java Platform	X	--	X	--
JSR-101: Java API for XML-Based RPC 1.1	Optional	--	--	--
JSR-67: Java APIs for XML Messaging 1.3	X	--	X	--
JSR-93: Java API for XML Registries	Optional	--	--	--
JSR-196: Java Authentication Service Provider Interface for Containers 1.1	X	--	X	--
JSR-115: Java Authorization Contract for Containers 1.5	X	--	X	--
JSR-88: Java EE Application Deployment 1.2	Optional	--	--	--
JSR-77: J2EE Management 1.1	X		X	
JSR-45: Debugging Support for Other Languages 1.0	X	X	X	X

Missing HornetQ and JMS?

The WildFly Web Profile doesn't include JMS (provided by HornetQ) by default. If you want to use messaging, make sure you start the server using the "Full Profile" configuration.

This document provides a quick overview on how to download and get started using WildFly 10 for your application development. For in-depth content on administrative features, refer to the WildFly 10 Admin Guide.

9.1.1 Download

WildFly 10 distributions can be obtained from:

wildfly.org/downloads

WildFly 10 provides a single distribution available in zip or tar file formats.

- **wildfly-10.0.0.Final.zip**
- **wildfly-10.0.0.Final.tar.gz**

9.1.2 Requirements

- Java SE 8 or later (we recommend that you use the latest update available)

9.1.3 Installation

Simply extract your chosen download to the directory of your choice. You can install WildFly 10 on any operating system that supports the zip or tar formats. Refer to the Release Notes for additional information related to the release.

9.1.4 WildFly - A Quick Tour

Now that you've downloaded WildFly 10, the next thing to discuss is the layout of the distribution and explore the server directory structure, key configuration files, log files, user deployments and so on. It's worth familiarizing yourself with the layout so that you'll be able to find your way around when it comes to deploying your own applications.

WildFly 10 Directory Structure

DIRECTORY	DESCRIPTION
appclient	Configuration files, deployment content, and writable areas used by the application client container run from this installation.
bin	Start up scripts, start up configuration files and various command line utilities like Vault, add-user and Java diagnostic report available for Unix and Windows environments
bin/client	Contains a client jar for use by non-maven based clients.
docs/schema	XML schema definition files
docs/examples/configs	Example configuration files representing specific use cases
domain	Configuration files, deployment content, and writable areas used by the domain mode processes run from this installation.
modules	WildFly 10 is based on a modular classloading architecture. The various modules used in the server are stored here.
standalone	Configuration files, deployment content, and writable areas used by the single standalone server run from this installation.
welcome-content	Default Welcome Page content

Standalone Directory Structure

In "***standalone***" mode each WildFly 10 server instance is an independent process (similar to previous JBoss AS versions; e.g., 3, 4, 5, or 6). The configuration files, deployment content and writable areas used by the single standalone server run from a WildFly installation are found in the following subdirectories under the top level "standalone" directory:

DIRECTORY	DESCRIPTION
configuration	Configuration files for the standalone server that runs off of this installation. All configuration information for the running server is located here and is the single place for configuration modifications for the standalone server.
data	Persistent information written by the server to survive a restart of the server
deployments	End user deployment content can be placed in this directory for automatic detection and deployment of that content into the server's runtime. NOTE: The server's management API is recommended for installing deployment content. File system based deployment scanning capabilities remain for developer convenience.
lib/ext	Location for installed library jars referenced by applications using the Extension-List mechanism
log	standalone server log files
tmp	location for temporary files written by the server
tmp/auth	Special location used to exchange authentication tokens with local clients so they can confirm that they are local to the running AS process.

Domain Directory Structure

A key feature of WildFly 10 is the managing multiple servers from a single control point. A collection of multiple servers are referred to as a "***domain***". Domains can span multiple physical (or virtual) machines with all WildFly instances on a given host under the control of a Host Controller process. The Host Controllers interact with the Domain Controller to control the lifecycle of the WildFly instances running on that host and to assist the Domain Controller in managing them. The configuration files, deployment content and writeable areas used by domain mode processes run from a WildFly installation are found in the following subdirectories under the top level "domain" directory:

DIRECTORY	DESCRIPTION
configuration	Configuration files for the domain and for the Host Controller and any servers running off of this installation. All configuration information for the servers managed wtihin the domain is located here and is the single place for configuration information.
content	an internal working area for the Host Controller that controls this installation. This is where it internally stores deployment content. This directory is not meant to be manipulated by end users. Note that "*domain*" mode does not support deploying content based on scanning a file system.
lib/ext	Location for installed library jars referenced by applications using the Extension-List mechanism
log	Location where the Host Controller process writes its logs. The Process Controller, a small lightweight process that actually spawns the other Host Controller process and any Application Server processes also writes a log here.
servers	Writable area used by each Application Server instance that runs from this installation. Each Application Server instance will have its own subdirectory, created when the server is first started. In each server's subdirectory there will be the following subdirectories: data -- information written by the server that needs to survive a restart of the server log -- the server's log files tmp -- location for temporary files written by the server
tmp	location for temporary files written by the server
tmp/auth	Special location used to exchange authentication tokens with local clients so they can confirm that they are local to the running AS process.

WildFly 10 Configurations

Standalone Server Configurations

- standalone.xml (*default*)
  - Java Enterprise Edition 7 web profile certified configuration with the required technologies plus those noted in the table above.
- standalone-ha.xml
  - Java Enterprise Edition 7 web profile certified configuration with high availability
- standalone-full.xml
  - Java Enterprise Edition 7 full profile certified configuration including all the required EE 7 technologies
- standalone-full-ha.xml
  - Java Enterprise Edition 7 full profile certified configuration with high availability

Domain Server Configurations

- domain.xml
  - Java Enterprise Edition 7 full and web profiles available with or without high availability

Important to note is that the ***domain*** and ***standalone*** modes determine how the servers are managed not what capabilities they provide.

Starting WildFly 10

To start WildFly 10 using the default web profile configuration in "*standalone*" mode, change directory to $JBOSS_HOME/bin.

```
./standalone.sh
```

To start the default web profile configuration using domain management capabilities,

```
./domain.sh
```

Starting WildFly 10 with an Alternate Configuration

If you choose to start your server with one of the other provided configurations, they can be accessed by passing the --server-config argument with the server-config file to be used.

To use the full profile with clustering capabilities, use the following syntax from $JBOSS_HOME/bin:

```
./standalone.sh --server-config=standalone-full-ha.xml
```

Similarly to start an alternate configuration in *domain* mode:

```
./domain.sh --domain-config=my-domain-configuration.xml
```

Alternatively, you can create your own selecting the additional subsystems you want to add, remove, or modify.

Test Your Installation

After executing one of the above commands, you should see output similar to what's shown below.

```
=========================================================================

  JBoss Bootstrap Environment

  JBOSS_HOME: /opt/wildfly-10.0.0.Final

  JAVA: java

  JAVA_OPTS:  -server -Xms64m -Xmx512m -XX:MetaspaceSize=96M -XX:MaxMetaspaceSize=256m
-Djava.net.preferIPv4Stack=true -Djboss.modules.system.pkgs=com.yourkit,org.jboss.byteman
-Djava.awt.headless=true

=========================================================================

11:46:11,161 INFO  [org.jboss.modules] (main) JBoss Modules version 1.5.1.Final
11:46:11,331 INFO  [org.jboss.msc] (main) JBoss MSC version 1.2.6.Final
11:46:11,391 INFO  [org.jboss.as] (MSC service thread 1-6) WFLYSRV0049: WildFly Full
10.0.0.Final (WildFly Core 2.0.10.Final) starting
<snip>
11:46:14,300 INFO  [org.jboss.as] (Controller Boot Thread) WFLYSRV0025: WildFly Full
10.0.0.Final (WildFly Core 2.0.10.Final) started in 1909ms - Started 267 of 553 services (371
services are lazy, passive or on-demand)
```

As with previous WildFly releases, you can point your browser to *http://localhost:8080* (if using the default configured http port) which brings you to the Welcome Screen:

From here you can access links to the WildFly community documentation set, stay up-to-date on the latest project information, have a discussion in the user forum and access the enhanced web-based Administration Console. Or, if you uncover a defect while using WildFly, report an issue to inform us (attached patches will be reviewed). This landing page is recommended for convenient access to information about WildFly 10 but can easily be replaced with your own if desired.

Managing your WildFly 10

WildFly 10 offers two administrative mechanisms for managing your running instance:

- web-based Administration Console
- command-line interface

Authentication

By default WildFly 10 is now distributed with security enabled for the management interfaces, this means that before you connect using the administration console or remotely using the CLI you will need to add a new user, this can be achieved simply by using the *add-user.sh* script in the bin folder.

After starting the script you will be guided through the process to add a new user: -

```
./add-user.sh
What type of user do you wish to add?
 a) Management User (mgmt-users.properties)
 b) Application User (application-users.properties)
(a):
```

In this case a new user is being added for the purpose of managing the servers so select option a.

You will then be prompted to enter the details of the new user being added: -

```
Enter the details of the new user to add.
Realm (ManagementRealm) :
Username :
Password :
Re-enter Password :
```

It is important to leave the name of the realm as 'ManagementRealm' as this needs to match the name used in the server's configuration, for the remaining fields enter the new username, password and password confirmation.

Provided there are no errors in the values entered you will then be asked to confirm that you want to add the user, the user will be written to the properties files used for authentication and a confirmation message will be displayed.

The modified time of the properties files are inspected at the time of authentication and the files reloaded if they have changed, for this reason you do not need to re-start the server after adding a new user.

Administration Console

To access the web-based Administration Console, simply follow the link from the Welcome Screen. To directly access the Management Console, point your browser at:

http://localhost:9990/console

NOTE: port 9990 is the default port configured.

```
<management-interfaces>
   <native-interface security-realm="ManagementRealm">
      <socket-binding native="management-native"/>
   </native-interface>
   <http-interface security-realm="ManagementRealm">
      <socket-binding http="management-http"/>
   </http-interface>
</management-interfaces>
```

If you modify the *management-http* socket binding in your running configuration: adjust the above command accordingly. If such modifications are made, then the link from the Welcome Screen will also be inaccessible.

If you have not yet added at least one management user an error page will be displayed asking you to add a new user, after a user has been added you can click on the 'Try Again' link at the bottom of the error page to try connecting to the administration console again.

Command-Line Interface

If you prefer to manage your server from the command line (or batching), the *jboss-cli.sh* script provides the same capabilities available via the web-based UI. This script is accessed from $JBOSS_HOME/bin directory; e.g.,

```
$JBOSS_HOME/bin/jboss-cli.sh --connect
Connected to standalone controller at localhost:9990
```

Notice if no host or port information provided, it will default to localhost:9990.

When running locally to the WildFly process the CLI will silently authenticate against the server by exchanging tokens on the file system, the purpose of this exchange is to verify that the client does have access to the local file system. If the CLI is connecting to a remote WildFly installation then you will be prompted to enter the username and password of a user already added to the realm.

Once connected you can add, modify, remove resources and deploy or undeploy applications. For a complete list of commands and command syntax, type ***help*** once connected.

Modifying the Example DataSource

As with previous JBoss application server releases, a default data source, ***ExampleDS***, is configured using the embedded H2 database for developer convenience. There are two ways to define datasource configurations:

1. as a module
2. as a deployment

In the provided configurations, H2 is configured as a module. The module is located in the $JBOSS_HOME/modules/com/h2database/h2 directory. The H2 datasource configuration is shown below.

```
<subsystem xmlns="urn:jboss:domain:datasources:1.0">
    <datasources>
        <datasource jndi-name="java:jboss/datasources/ExampleDS" pool-name="ExampleDS">
            <connection-url>jdbc:h2:mem:test;DB_CLOSE_DELAY=-1</connection-url>
            <driver>h2</driver>
            <pool>
                <min-pool-size>10</min-pool-size>
                <max-pool-size>20</max-pool-size>
                <prefill>true</prefill>
            </pool>
            <security>
                <user-name>sa</user-name>
                <password>sa</password>
            </security>
        </datasource>
        <xa-datasource jndi-name="java:jboss/datasources/ExampleXADS" pool-name="ExampleXADS">
           <driver>h2</driver>
           <xa-datasource-property name="URL">jdbc:h2:mem:test</xa-datasource-property>
           <xa-pool>
                <min-pool-size>10</min-pool-size>
                <max-pool-size>20</max-pool-size>
                <prefill>true</prefill>
           </xa-pool>
           <security>
                <user-name>sa</user-name>
                <password>sa</password>
           </security>
        </xa-datasource>
        <drivers>
            <driver name="h2" module="com.h2database.h2">
                <xa-datasource-class>org.h2.jdbcx.JdbcDataSource</xa-datasource-class>
            </driver>
        </drivers>
  </datasources>
</subsystem>
```

The datasource subsystem is provided by the IronJacamar project. For a detailed description of the available configuration properties, please consult the project documentation.

- IronJacamar homepage: http://www.jboss.org/ironjacamar
- Project Documentation: http://www.jboss.org/ironjacamar/docs
- Schema description: http://docs.jboss.org/ironjacamar/userguide/1.0/en-US/html/deployment.html#deployingds_descriptor

Configure Logging in WildFly 10

WildFly 10 logging can be configured with the web console or the command line interface. You can get more detail on the Logging Configuration page.

Turn on debugging for a specific category with CLI:

```
/subsystem=logging/logger=org.jboss.as:add(level=DEBUG)
```

By default the `server.log` is configured to include all levels in it's log output. In the above example we changed the console to also display debug messages.

9.2 JavaEE 6 Tutorial

Coming Soon

This guide is still under development, check back soon!

9.2.1 Standard JavaEE 6 Technologies

1. Enterprise JavaBeans Technology (EJB)
2. Java Servlet Technology
3. Java Server Faces Technology (JSF)
4. Java Persistence API (JPA)
5. Java Transaction API (JTA)
6. Java API for RESTful Web Services (JAX-RS)
7. Java API for XML Web Services (JAX-WS)
8. Managed Beans
9. Contexts and Dependency Injection (CDI)
10. Bean Validation
11. Java Message Service API (JMS)
12. JavaEE Connector Architecture (JCA)
13. JavaMail API
14. Java Authorization Contract for Containers (JACC)
15. Java Authentication Service Provider Interface for Containers (JASPIC)

9.2.2 JBoss AS7 Extension Technologies

1. OSGi Technology
2. Management Interface

9.2.3 Standard JavaEE 6 Technologies

Coming Soon

This guide is still under development, check back soon!

1. Enterprise JavaBeans Technology (EJB)
2. Java Servlet Technology
3. Java Server Faces Technology (JSF)
4. Java Persistence API (JPA)
5. Java Transaction API (JTA)
6. Java API for RESTful Web Services (JAX-RS)
7. Java API for XML Web Services (JAX-WS)
8. Managed Beans
9. Contexts and Dependency Injection (CDI)
10. Bean Validation
11. Java Message Service API (JMS)
12. JavaEE Connector Architecture (JCA)
13. JavaMail API
14. Java Authorization Contract for Containers (JACC)
15. Java Authentication Service Provider Interface for Containers (JASPIC)

Java API for RESTful Web Services (JAX-RS)

Content

Tutorial Overview

This chapter describes the Java API for RESTful web services (JAX-RS, defined in JSR331). RESTEasy is an portable implementation of this specification which can run in any Servlet container. Tight integration with JBoss Application Server is available for optimal user experience in that environment. While JAX-RS is only a server-side specification, RESTeasy has innovated to bring JAX-RS to the client through the RESTEasy JAX-RS Client Framework.

Detailed documentation on RESTEasy is available here.
The source for this tutorial is in github repository git://github.com/tdiesler/javaee-tutorial.git

OpenShift, is a portfolio of portable cloud services for deploying and managing applications in the cloud. This tutorial shows how to deploy a RESTful web service on the free OpenShift Express JavaEE cartridge that runs JBossAS 7.

An application running on Android shows how to leverage JBoss technology on mobile devices. Specifically, we show how use the RESTEasy client API from an Android device to integrate with a RESTful service running on a JBossAS 7 instance in the cloud.

The following topics are addressed

- What are RESTful web services
- Creating a RESTful server endpoint
- Deploying a RESTful endpoint to a JBossAS instance in the cloud
- RESTEasy client running on an Android mobile device

What are RESTful Web Services?

Coming Soon

This section is still under development.

RESTful web services are designed to expose APIs on the web. REST stands for **Re**presentational **S**tate **T**ransfer. It aims to provide better performance, scalability, and flexibility than traditinoal web services, by allowing clients to access data and resources using predictable URLs. Many well-known public web services expose RESTful APIs.

The Java 6 Enterprise Edition specification for RESTful services is JAX-RS. It is covered by JSR-311 (http://jcp.org/jsr/detail/311.jsp). In the REST model, the server exposes APIs through specific URIs (typically URLs), and clients access those URIs to query or modify data. REST uses a stateless communication protocol. Typically, this is HTTP.

The following is a summary of RESTful design principles:

- A URL is tied to a resource using the `@Path` annotation. Clients access the resource using the URL.
- Create, Read, Update, and Delete (CRUD) operations are accessed via `PUT`, `GET`, `POST`, and `DELETE` requests in the HTTP protocol.
  - `PUT` creates a new resource.
  - `DELETE` deletes a resource.
  - `GET` retrieves the current state of a resource.
  - `POST` updates a resources's state.
- Resources are decoupled from their representation, so that clients can request the data in a variety of different formats.
- Stateful interactions require explicit state transfer, in the form of URL rewriting, cookies, and hidden form fields. State can also be embedded in response messages.

Creating a RESTful endpoint

A RESTful endpoint is deployed as JavaEE web archive (WAR). For this tutorial we use a simple library application to manage some books. There are two classes in this application:

- Library
- Book

The Book is a plain old Java object (POJO) with two attributes. This is a simple Java representation of a RESTful entity.

```
public class Book {

    private String isbn;
    private String title;

    ...
}
```

The Library is the RESTful Root Resource. Here we use a set of standard JAX-RS annotations to define

- The root path to the library resource
- The wire representation of the data (MIME type)
- The Http methods and corresponding paths

```
@Path("/library")
@Consumes({ "application/json" })
@Produces({ "application/json" })
public class Library {

    @GET
    @Path("/books")
    public Collection<Book> getBooks() {
    ...
    }

    @GET
    @Path("/book/{isbn}")
    public Book getBook(@PathParam("isbn") String id) {
    ...
    }

    @PUT
    @Path("/book/{isbn}")
    public Book addBook(@PathParam("isbn") String id, @QueryParam("title") String title) {
    ...
    }

    @POST
    @Path("/book/{isbn}")
    public Book updateBook(@PathParam("isbn") String id, String title) {
    ...
    }

    @DELETE
    @Path("/book/{isbn}")
    public Book removeBook(@PathParam("isbn") String id) {
    ...
    }
}
```

The Library root resource uses these JAX-RS annotations:

Annotation	Description
@Path	Identifies the URI path that a resource class or class method will serve requests for
@Consumes	Defines the media types that the methods of a resource class can accept
@Produces	Defines the media type(s) that the methods of a resource class can produce
@GET	Indicates that the annotated method responds to HTTP GET requests
@PUT	Indicates that the annotated method responds to HTTP PUT requests
@POST	Indicates that the annotated method responds to HTTP POST requests
@DELETE	Indicates that the annotated method responds to HTTP DELETE requests

For a full description of the available JAX-RS annotations, see the JAX-RS API documentation.

Package and build the endpoint

To package the endpoint we create a simple web archive and include a web.xml with the following content

Review

AS7-1674 Remove or explain why web.xml is needed for RESTful endpoints

```
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
    <servlet-mapping>
        <servlet-name>javax.ws.rs.core.Application</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>
```

The root context is defined in jboss-web.xml

```
<jboss-web>
  <context-root>jaxrs-sample</context-root>
</jboss-web>
```

The code for the JAX-RS part of this tutorial is available on https://github.com/tdiesler/javaee-tutorial/tree/master/jaxrs. In this step we clone the repository and build the endpoint using maven. There are a number of JAX-RS client tests that run against a local JBossAS 7 instance. Before we build the project, we set the JBOSS_HOME environment variable accordingly.

Arquillian, the test framework we use throughout this tutorial, can manage server startup/shutdown. It is however also possible to startup the server instance manually before you run the tests. The latter allows you to look at the console and see what log output the deployment phase and JAX-RS endpoint invocations produce.

```
$ git clone git://github.com/tdiesler/javaee-tutorial.git
Cloning into javaee-tutorial...

$ cd javaee-tutorial/jaxrs
$ export JBOSS_HOME=~/workspace/jboss-as-7.0.1.Final
$ mvn install
...
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary:
[INFO]
[INFO] JavaEE Tutorial - JAX-RS ......................... SUCCESS [1.694s]
[INFO] JavaEE Tutorial - JAX-RS Server .................. SUCCESS [2.392s]
[INFO] JavaEE Tutorial - JAX-RS Client .................. SUCCESS [7.304s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 12.142s
```

Deploy the endpoint to OpenShift

First we need to create a free OpenShift Express account and select the JavaEE cartridge that runs JBossAS 7. Once we have received the confirmation email from OpenShift we can continue to create our subdomain and deploy the RESTful endpoint. A series of videos on the OpenShift Express page shows you how to do this. There is also an excellent quick start document that you have access to after login.

For this tutorial we assume you have done the above and that we can continue by creating the OpenShift application. This step sets up your JBossAS 7 instance in the cloud. Additionally a Git repository is configured that gives access to your deployed application.

```
$ rhc-create-app -a tutorial -t jbossas-7.0
Password:

Attempting to create remote application space: tutorial
Successfully created application: tutorial
Now your new domain name is being propagated worldwide (this might take a minute)...

Success!  Your application is now published here:

      http://tutorial-tdiesler.rhcloud.com/

The remote repository is located here:

    ssh://79dcb9db5e134cccb9d1ba33e6089667@tutorial-tdiesler.rhcloud.com/~/git/tutorial.git/
```

Next, we can clone the remote Git repository to our local workspace

```
$ git clone
ssh://79dcb9db5e134cccb9d1ba33e6089667@tutorial-tdiesler.rhcloud.com/~/git/tutorial.git
Cloning into tutorial...
remote: Counting objects: 24, done.
remote: Compressing objects: 100% (14/14), done.
remote: Total 24 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (24/24), 21.84 KiB, done.

ls -1 tutorial
deployments
pom.xml
README
src
```

Because we want to deploy an already existing web application, which we'll build in the next step, we can safely remove the source artefacts from the repository.

```
$ rm -rf tutorial/src tutorial/pom.xml
```

Now we copy the JAX-RS endpoint webapp that we build above to the 'deployments' folder and commit the changes.

```
$ cp javaee-tutorial/jaxrs/server/target/javaee-tutorial-jaxrs-server-1.0.0-SNAPSHOT.war
tutorial/deployments
$ cd tutorial; git commit -a -m "Initial jaxrs endpoint deployment"
[master be5b5a3] Initial jaxrs endpoint deployment
 7 files changed, 0 insertions(+), 672 deletions(-)
 create mode 100644 deployments/javaee-tutorial-jaxrs-server-1.0.0-SNAPSHOT.war
 delete mode 100644 pom.xml
 delete mode 100644 src/main/java/.gitkeep
 delete mode 100644 src/main/resources/.gitkeep
 delete mode 100644 src/main/webapp/WEB-INF/web.xml
 delete mode 100644 src/main/webapp/health.jsp
 delete mode 100644 src/main/webapp/images/jbosscorp_logo.png
 delete mode 100644 src/main/webapp/index.html
 delete mode 100644 src/main/webapp/snoop.jsp

$ git push origin
Counting objects: 6, done.
...
remote: Starting application...Done
```

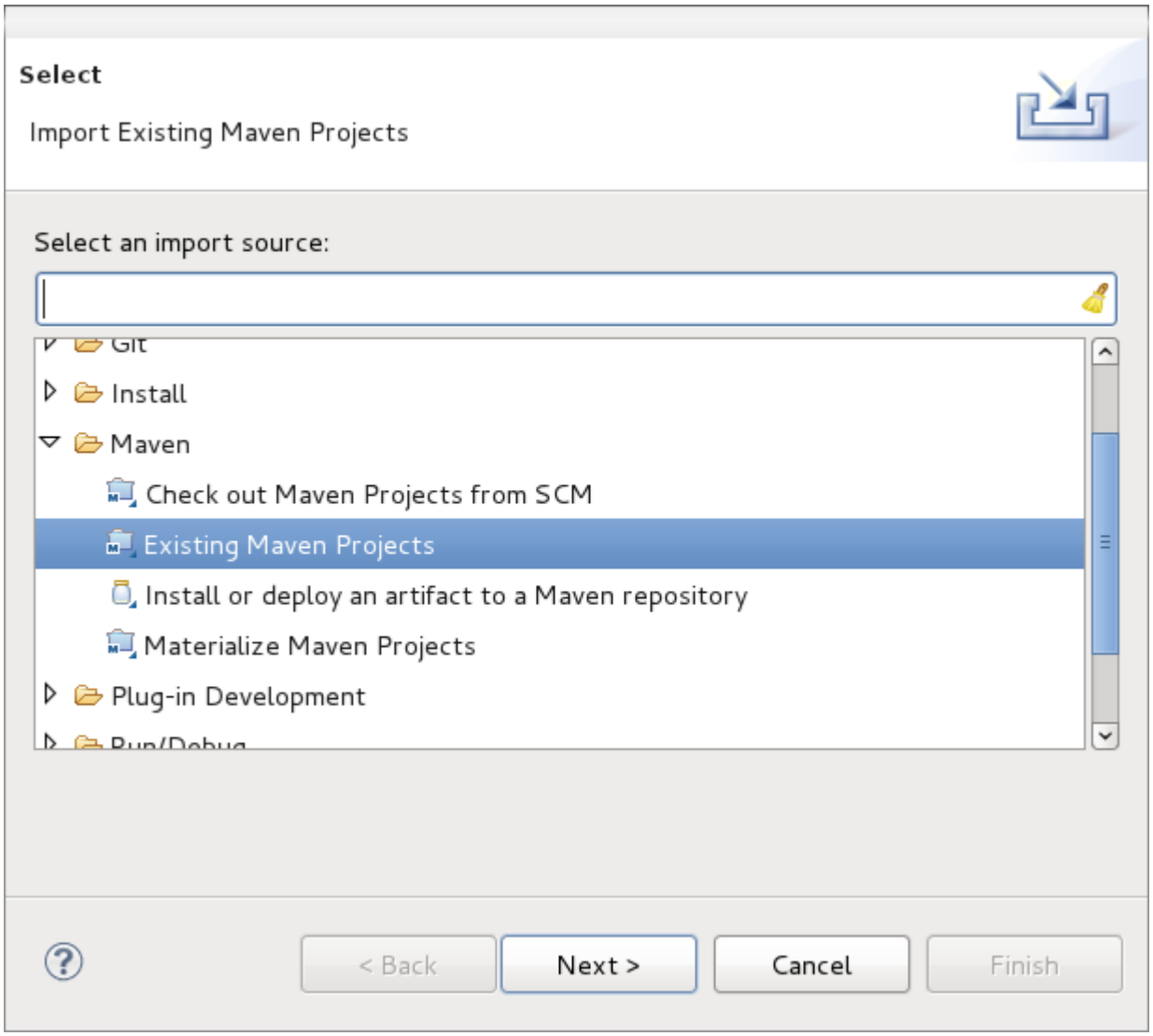
You can now use curl or your browser to see the JAX-RS endpoint in action. The following URL lists the books that are currently registered in the library.

```
$ curl http://tutorial-tdiesler.rhcloud.com/jaxrs-sample/library/books
[
{"title":"The Judgment","isbn":"001"},
{"title":"The Stoker","isbn":"002"},
{"title":"Jackals and Arabs","isbn":"003"},
{"title":"The Refusal","isbn":"004"}
]
```

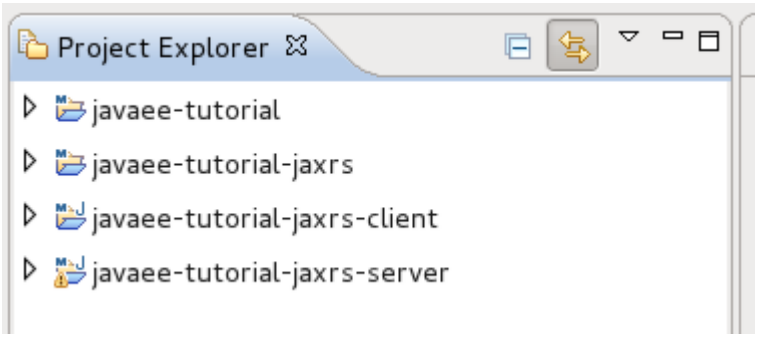
Building the mobile client

The source associated with this tutorial contains a fully working mobile client application for the Android framework. If not done so already please follow steps described in Installing the SDK. In addition to the Android SDK, I recommend installing the m2eclipse and the EGit plugin to Eclipse.

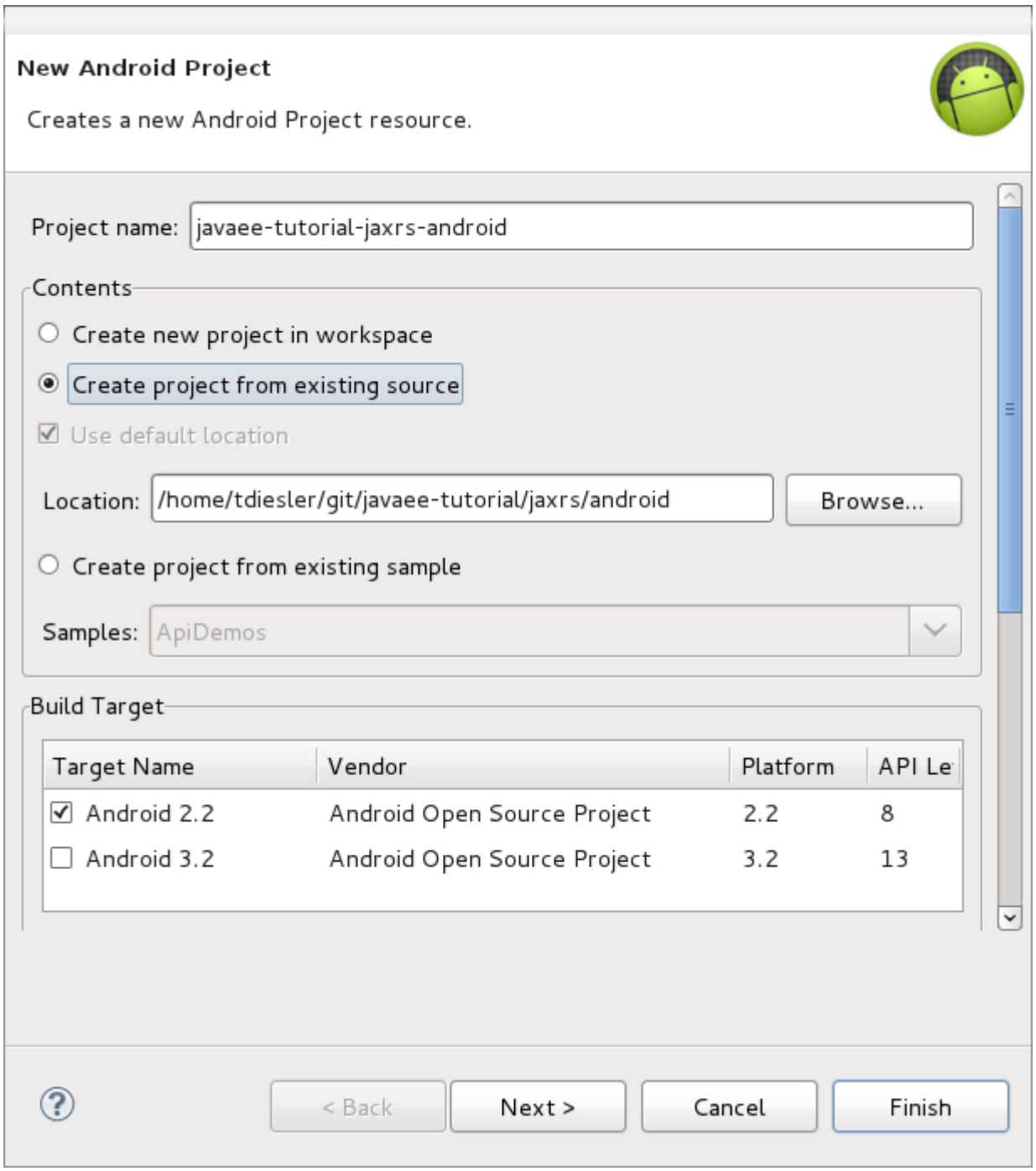
First, go to File|Import... and choose "Existing Maven Projects" to import the tutorial sources



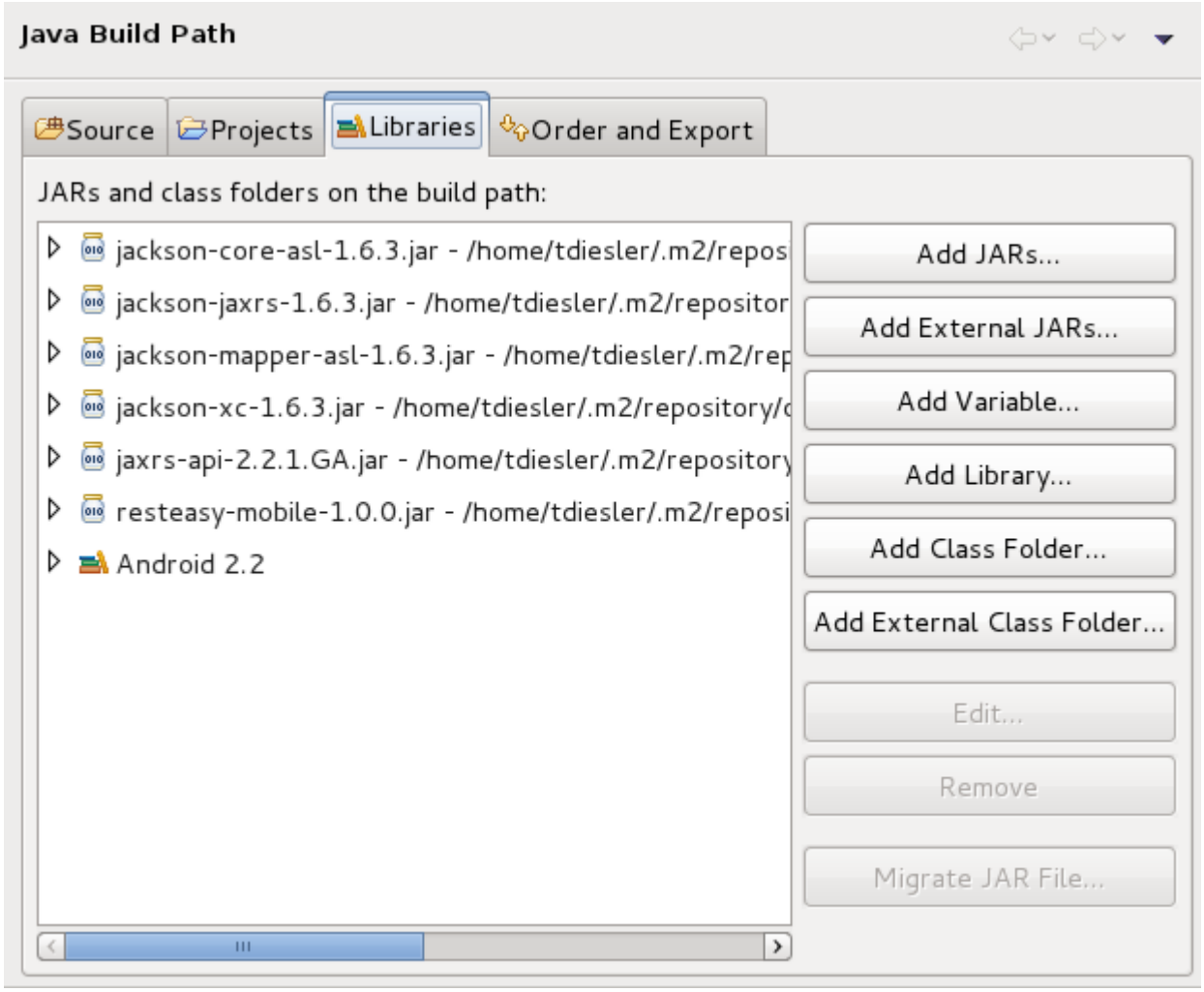
You project view should look like this



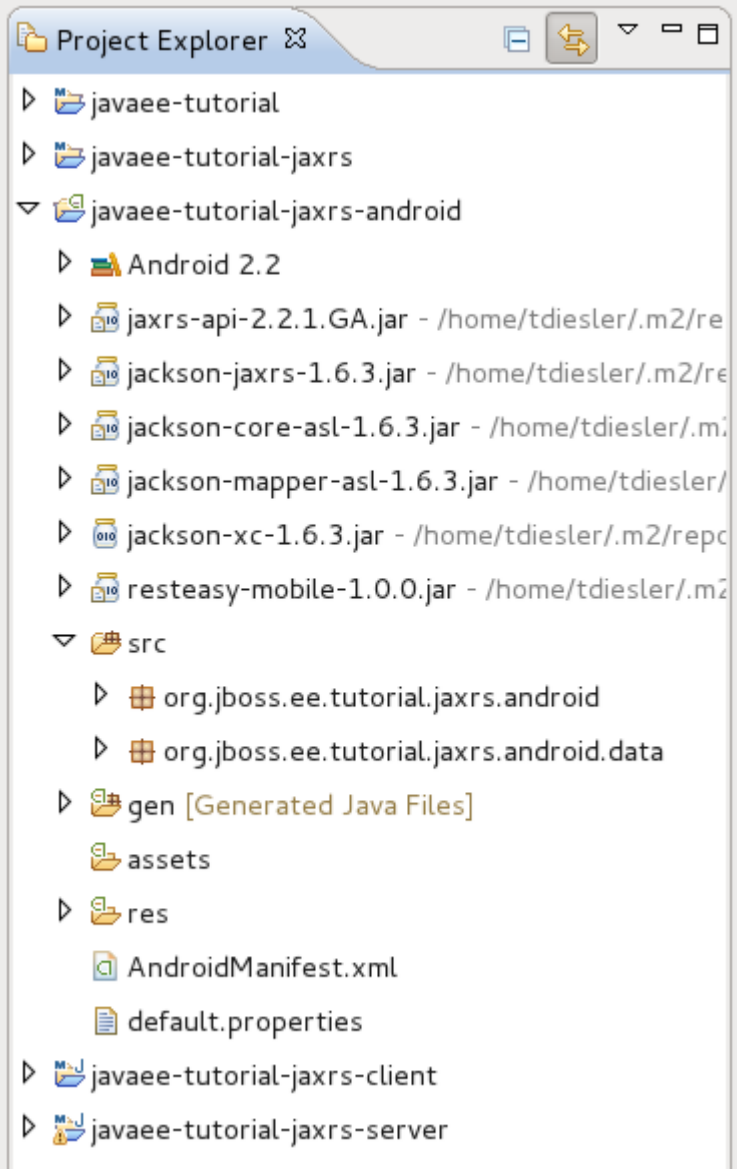
Then go to File|New|Android Project and fill out the first wizard page like this



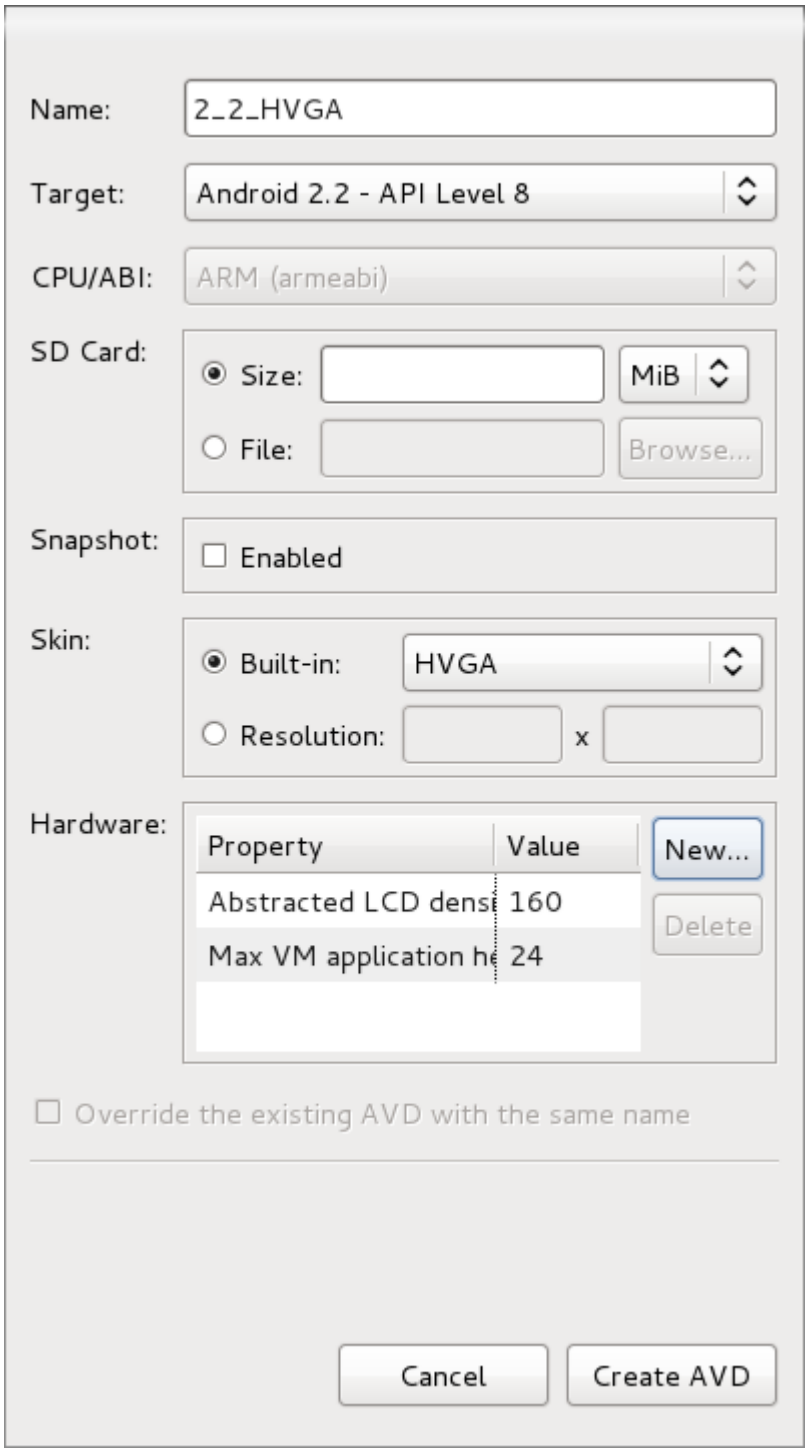
Click Finish. Next, go to Project|Properties|Build Path|Libraries and add these external libraries to your android project.



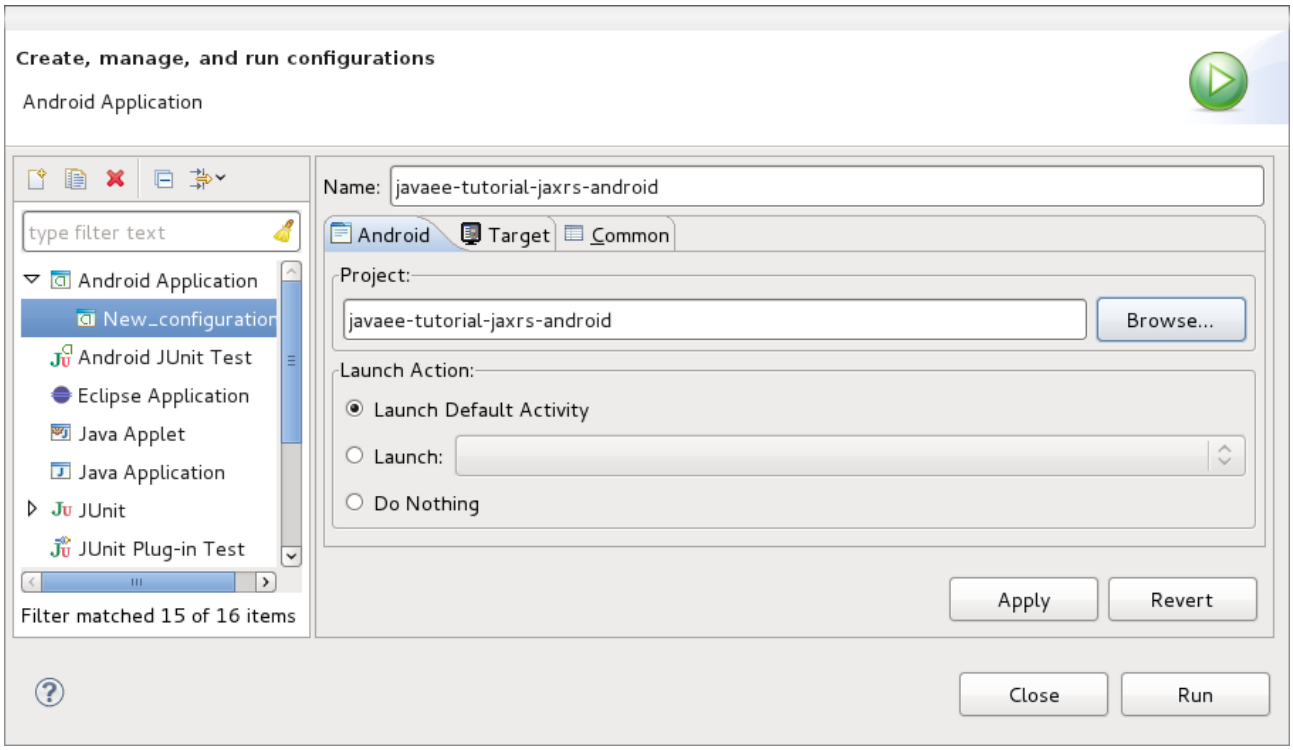
You final project view should look like this



To run the application in the emulator, we need an Android Virtual Device (AVD). Go to Window|Android SDK and AVD Manager and create a new AVD like this



Now go to Run|Configuration to create a new run configuration for the client app.



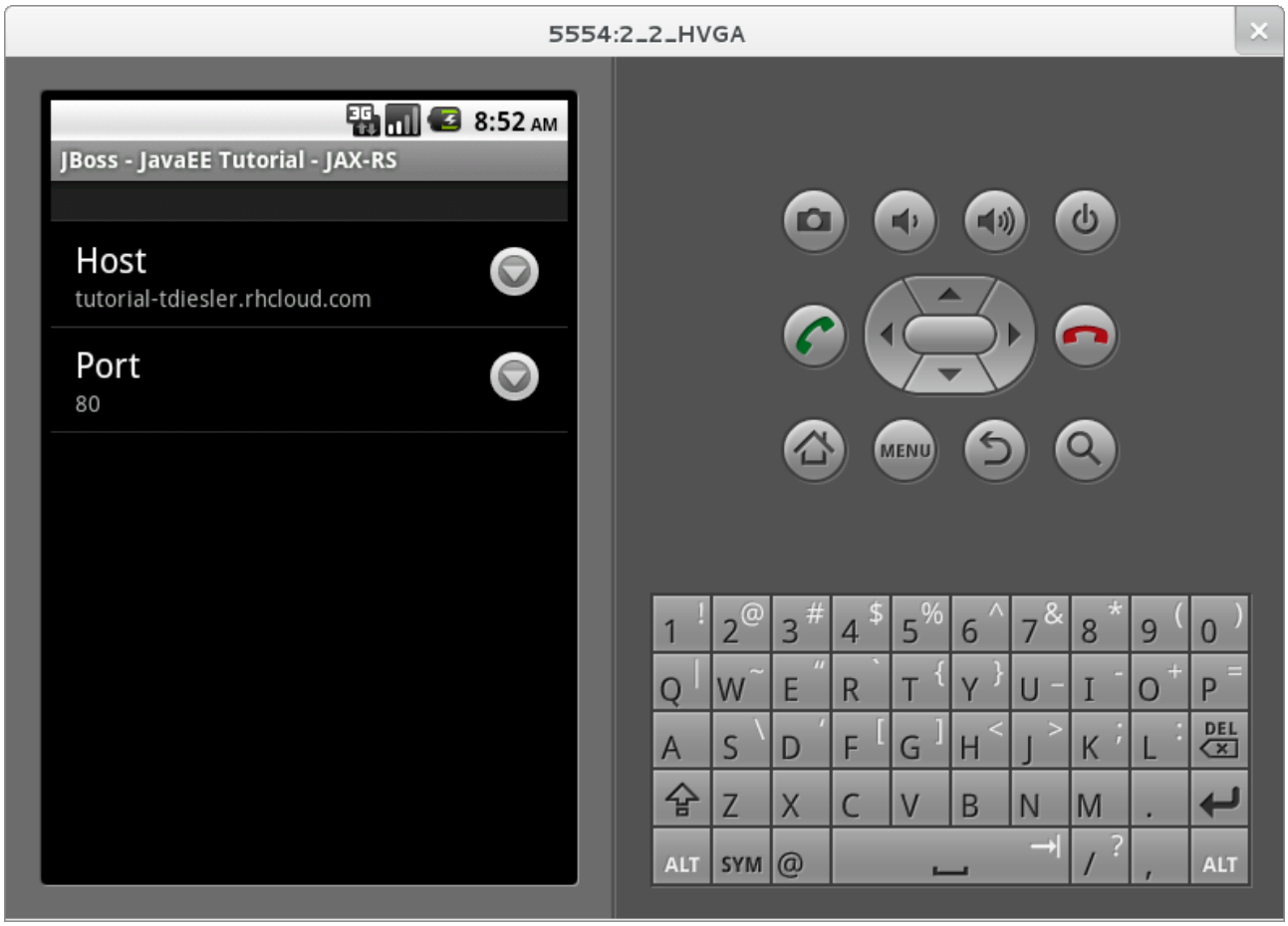
Now you should be able to launch the application in the debugger. Right click on the javaee-tutorial-jaxrs-android project and select Debug As|Android Application. This should launch the emulator, which now goes though a series of boot screens until it eventually displays the Android home screen. This will take a minute or two if you do this for the first time.


5554:2_2_HVGA
ANDROID
MENU
ALT
SYM
ALT

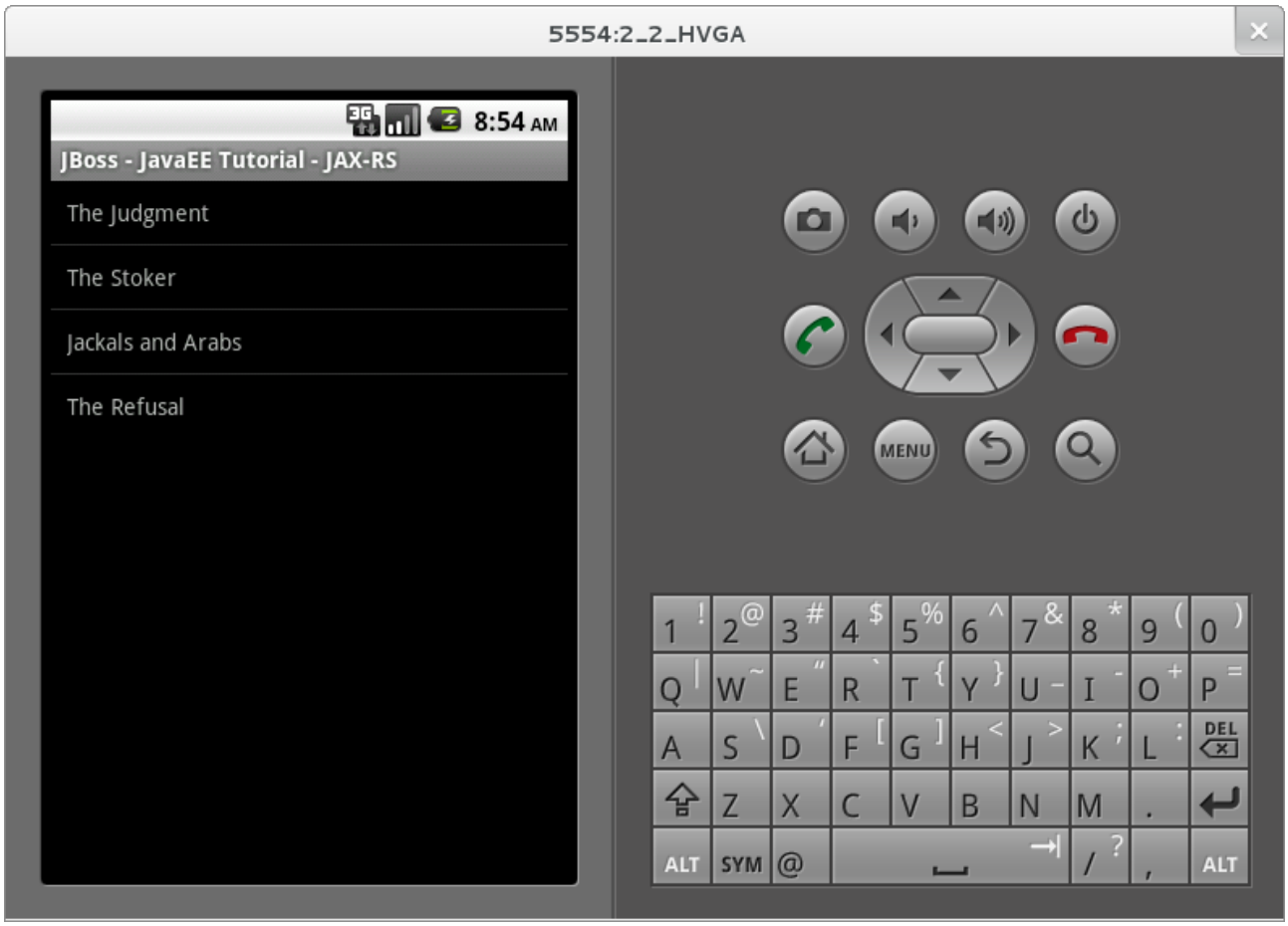

5554:2_2_HVGA
ANDROID
MENU
ALT
SYM
ALT

When you unlock the home screen by dragging the little green lock to the right. You should see the the running JAX-RS client application.

Finally, you need to configure the host that the client app connects to. This would be the same as you used above to curl the library list. In the emulator click Menu|Host Settings and enter the host address of your OpenShift application.



When going back to the application using the little back arrow next to Menu, you should see a list of books.



You can now add, edit and delete books and switch between your browser and the emulator to verify that the client app is not cheating and that the books are in fact in the cloud on your JBossAS 7 instance.

In Eclipse you can go to the Debug perspective and click on the little Android robot in the lower right corner. This will display the LogCat view, which should display log output from that Android system as well as from this client app

```
08-30 09:05:46.180: INFO/JaxrsSample(269): removeBook: Book [isbn=1234, title=1234]
08-30 09:05:46.210: INFO/JaxrsSample(269): requestURI:
http://tutorial-tdiesler.rhcloud.com:80/jaxrs-sample/library
08-30 09:05:46.860: INFO/global(269): Default buffer size used in BufferedInputStream
constructor. It would be better to be explicit if an 8k buffer is required.
08-30 09:05:46.920: INFO/JaxrsSample(269): getBooks: [Book [isbn=001, title=The Judgment], Book
[isbn=002, title=The Stoker], Book [isbn=003, title=Jackals and Arabs], Book [isbn=004,
title=The Refusal]]
```

Exploring the mobile client

There is a lot to writing high quality mobile applications. The goal of this little application is to get you started with JBossAS 7 / Android integration. There is also a portable approach to writing mobile applications. A popular one would be through PhoneGap. With PhoneGap you write your application in HTML+CSS+Java Script. It then runs in the browser of your mobile device. Naturally, not the full set of mobile platform APIs would be available through this approach.

The JAX-RS client application uses an annotated library client interface

```
@Consumes({ "application/json" })
@Produces({ "application/json" })
public interface LibraryClient {

    @GET
    @Path("/books")
    public List<Book> getBooks();

    @GET
    @Path("/book/{isbn}")
    public Book getBook(@PathParam("isbn") String id);

    @PUT
    @Path("/book/{isbn}")
    public Book addBook(@PathParam("isbn") String id, @QueryParam("title") String title);

    @POST
    @Path("/book/{isbn}")
    public Book updateBook(@PathParam("isbn") String id, String title);

    @DELETE
    @Path("/book/{isbn}")
    public Book removeBook(@PathParam("isbn") String id);
}
```

There are two implementations of this interface available.

- LibraryHttpclient
- LibraryResteasyClient

The first uses APIs that are available in the Android SDK natively. The code is much more involved, but there would be no need to add external libraries (i.e. resteasy, jackson, etc). The effect is that the total size of the application is considerably smaller in size (i.e. 40k)

```
@Override
    public List<Book> getBooks() {
        List<Book> result = new ArrayList<Book>();
        String content = get("books");
        Log.d(LOG_TAG, "Result content:" + content);
        if (content != null) {
            try {
                JSONTokener tokener = new JSONTokener(content);
                JSONArray array = (JSONArray) tokener.nextValue();
                for (int i = 0; i < array.length(); i++) {
                    JSONObject obj = array.getJSONObject(i);
                    String title = obj.getString("title");
                    String isbn = obj.getString("isbn");
                    result.add(new Book(isbn, title));
                }
            } catch (JSONException ex) {
                ex.printStackTrace();
            }
        }
        Log.i(LOG_TAG, "getBooks: " + result);
        return result;
    }

    private String get(String path) {
        try {
            HttpGet request = new HttpGet(getRequestURI(path));
            HttpResponse res = httpClient.execute(request);
            String content = EntityUtils.toString(res.getEntity());
            return content;
        } catch (Exception ex) {
            ex.printStackTrace();
            return null;
        }
    }
```

The second implementation uses the fabulous RESTEasy client proxy to interact with the JAX-RS endpoint. The details of Http connectivity and JSON data binding is transparently handled by RESTEasy. The total size of the application is considerably bigger in size (i.e. 400k)

```
@Override
    public List<Book> getBooks() {
        List<Book> result = new ArrayList<Book>();
        try {
            result = getLibraryClient().getBooks();
        } catch (RuntimeException ex) {
            ex.printStackTrace();
        }
        Log.i(LOG_TAG, "getBooks: " + result);
        return result;
    }
```

Stay tuned for an update on a much more optimized version of the RESTEasy mobile client. Feasible is also a RESTEasy JavaScript library that would enable the portable PhoneGap approach.

Java Servlet Technology

Coming Soon

This guide is still under development, check back soon!

Content

- Asynchronous Support

Asynchronous Support

Java Server Faces Technology (JSF)

Java Persistence API (JPA)

Java Transaction API (JTA)

Coming Soon

This guide is still under development, check back soon!

Managed Beans

Contexts and Dependency Injection (CDI)

Bean Validation

Java Message Service API (JMS)

Coming Soon

This guide is still under development, check back soon!

- Configure JBossAS for Messaging
- Adding the message destinations

Configure JBossAS for Messaging

Currently, the default configuration does not include the JMS subsystem. To enable JMS in the standalone server you need to add these configuration items to standalone.xml or simply use standalone-full.xml.

```
<extension module="org.jboss.as.messaging"/>

<subsystem xmlns="urn:jboss:domain:messaging:1.0">
    <!-- Default journal file size is 10Mb, reduced here to 100k for faster first boot -->
    <journal-file-size>102400</journal-file-size>
    <journal-min-files>2</journal-min-files>
    <journal-type>NIO</journal-type>
    <!-- disable messaging persistence -->
    <persistence-enabled>false</persistence-enabled>

    <connectors>
       <netty-connector name="netty" socket-binding="messaging" />
       <netty-connector name="netty-throughput" socket-binding="messaging-throughput">
          <param key="batch-delay" value="50"/>
       </netty-connector>
       <in-vm-connector name="in-vm" server-id="0" />
    </connectors>

    <acceptors>
       <netty-acceptor name="netty" socket-binding="messaging" />
       <netty-acceptor name="netty-throughput" socket-binding="messaging-throughput">
          <param key="batch-delay" value="50"/>
          <param key="direct-deliver" value="false"/>
       </netty-acceptor>
        <acceptor name="stomp-acceptor">

<factory-class>org.hornetq.core.remoting.impl.netty.NettyAcceptorFactory</factory-class>
         <param key="protocol" value="stomp" />
         <param key="port" value="61613" />
       </acceptor>
       <in-vm-acceptor name="in-vm" server-id="0" />
    </acceptors>

    <security-settings>
       <security-setting match="#">
           <permission type="createNonDurableQueue" roles="guest"/>
           <permission type="deleteNonDurableQueue" roles="guest"/>
            <permission type="consume" roles="guest"/>
            <permission type="send" roles="guest"/>
       </security-setting>
    </security-settings>

    <address-settings>
       <!--default for catch all-->
       <address-setting match="#">
          <dead-letter-address>jms.queue.DLQ</dead-letter-address>
          <expiry-address>jms.queue.ExpiryQueue</expiry-address>
          <redelivery-delay>0</redelivery-delay>
          <max-size-bytes>10485760</max-size-bytes>
          <message-counter-history-day-limit>10</message-counter-history-day-limit>
          <address-full-policy>BLOCK</address-full-policy>
```

```
       </address-setting>
    </address-settings>

    <!--JMS Stuff-->
   <jms-connection-factories>
      <connection-factory name="InVmConnectionFactory">
         <connectors>
            <connector-ref connector-name="in-vm"/>
         </connectors>
         <entries>
            <entry name="java:/ConnectionFactory"/>
         </entries>
      </connection-factory>
      <connection-factory name="RemoteConnectionFactory">
         <connectors>
            <connector-ref connector-name="netty"/>
         </connectors>
         <entries>
            <entry name="RemoteConnectionFactory"/>
         </entries>
      </connection-factory>
      <pooled-connection-factory name="hornetq-ra">
         <transaction mode="xa"/>
         <connectors>
            <connector-ref connector-name="in-vm"/>
         </connectors>
         <entries>
            <entry name="java:/JmsXA"/>
            <!-- Global JNDI entry used to provide a default JMS Connection factory to EE
application -->
            <entry name="java:jboss/DefaultJMSConnectionFactory"/>
         </entries>
      </pooled-connection-factory>
   </jms-connection-factories>

   <jms-destinations>
      <jms-queue name="testQueue">
         <entry name="queue/test"/>
      </jms-queue>
      <jms-topic name="testTopic">
         <entry name="topic/test"/>
      </jms-topic>
   </jms-destinations>
</subsystem>

<socket-binding name="messaging" port="5445" />
<socket-binding name="messaging-throughput" port="5455"/>
```

Alternatively run the server using the preview configuration

```
$ bin/standalone.sh --server-config=standalone-preview.xml
```

Adding the message destinations

For this tutorial we use two message destinations

- BidQueue - The queue that receives the client bids
- AuctionTopic - The topic that publishes the start of a new auction

You can either add the message destinations by using the Command Line Interface (CLI)

```
$ bin/jboss-admin.sh --connect
Connected to standalone controller at localhost:9999
[standalone@localhost:9999 /] jms-queue add --queue-address=BidQueue --entries=queue/bid
[standalone@localhost:9999 /] jms-topic add --topic-address=AuctionTopic --entries=topic/auction
[standalone@localhost:9999 /] exit
Closed connection to localhost:9999
```

or by adding them to the subsytem configuration as shown above.

JavaEE Connector Architecture (JCA)

JavaMail API

Java Authorization Contract for Containers (JACC)

In order to register your own JACC Module, you'll need to create a server module containing the required classes, and then set three system properties for WildFly to take it. Such a module would depend on the "javax.api" and "javaee.api" modules.

An example module.xml for such a module could be:

```
<module xmlns="urn:jboss:module:1.1" name="com.example.customjacc">
  <resources>
    <resource-root path="customjacc.jar"/>
  </resources>
  <dependencies>
    <module name="javax.api"/>
    <module name="javaee.api"/>
  </dependencies>
</module>
```

The specified JAR needs to contain at least two classes, as mandated by the JACC spec:

- A `PolicyProvider` implementation: in our example, it'll be `com.example.customjacc.CustomPolicy.`
- A `PolicyConfigurationFactory` implementation: `com.example.customjacc.CustomPolicyConfigurationFactory` in our case.

The spec requires two system properties to be set for the server to register the JACC Module.

For a server running in standalone mode, put the following commands in the JBoss CLI:

```
[standalone@localhost:9990 /]
/system-property=javax.security.jacc.policy.provider:add(value=com.example.customjacc.CustomPolicy
/]
/system-property=javax.security.jacc.PolicyConfigurationFactory.provider:add(value=com.example.cus
```

Another property is needed to make WildFly know where to load the classes from:

```
[standalone@localhost:9990 /]
/system-property=org.jboss.as.security.jacc-module:add(value=com.example.customjacc)
```

Java Authentication Service Provider Interface for Containers (JASPIC)

JASPI is not available by default for deployments, and a specific Security Domain must be created to use it. For a simplified developer experience, a default JASPI Domain is already bundled, called `jaspitest`.

To make use of it, a Web Application only needs to specifiy the desired security domain in the `jboss-web.xml` deployment descriptor. This file should be located under the `WEB-INF` directory. An example `jboss-web.xml` enabling the default JASPI domain:

```
<?xml version="1.0"?>
<jboss-web xmlns="http://www.jboss.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee
http://www.jboss.org/j2ee/schema/jboss-web_10_0.xsd"
    version="10.0">
    <security-domain>jaspitest</security-domain>
</jboss-web>
```

For EAR deployments, a `jboss-app.xml` like the following should be used instead, placed under the root META-INF directory:

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss-app xmlns="http://www.jboss.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    version="7.0">
    <security-domain>jaspitest</security-domain>
</jboss-app>
```

Enterprise JavaBeans Technology (EJB)

In this section we'll look at the two main types of beans (Session Beans and Message Driven Beans), the methods to access and the packaging possibilities of beans.

Coming Soon

This guide is still under development, check back soon!

Session Beans

Stateful Session Beans

Stateless Session Beans

Singleton Session Beans

Message Driven Beans

How can Enterprise JavaBeans can be accessed?

Remote call invocation

Local call invocation

Web Services

Packaging

Java API for XML Web Services (JAX-WS)

JBossWS uses the JBoss Application Server as its target container. The following examples focus on web service deployments that leverage EJB3 service implementations and the JAX-WS programming models. For further information on POJO service implementations and advanced topics you need consult the user guide.

Developing web service implementations

JAX-WS does leverage annotations in order to express web service meta data on Java components and to describe the mapping between Java data types and XML. When developing web service implementations you need to decide whether you are going to start with an abstract contract (WSDL) or a Java component.

If you are in charge to provide the service implementation, then you are probably going to start with the implementation and derive the abstract contract from it. You are probably not even getting in touch with the WSDL unless you hand it to 3rd party clients. For this reason we are going to look at a service implementation that leverages JSR-181 annotations.

Even though detailed knowledge of web service meta data is not required, it will definitely help if you make yourself familiar with it. For further information see

- Web service meta data (JSR-181)
- Java API for XML binding (JAXB)
- Java API for XML-Based Web Services

The service implementation class

When starting from Java you must provide the service implementation. A valid endpoint implementation class must meet the following requirements:

- It *must* carry a `javax.jws.WebService` annotation (see JSR 181)
- All method parameters and return types *must* be compatible with the JAXB 2.0

Let's look at a sample EJB3 component that is going to be exposed as a web service.

Don't be confused with the EJB3 annotation `@Stateless`. We concentrate on the `@WebService` annotation for now.

Implementing the service

```
package org.jboss.test.ws.jaxws.samples.retail.profile;

import javax.ejb.Stateless;
import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.soap.SOAPBinding;

@Stateless                                                          (1)
@WebService(                                                        (2)
   name="ProfileMgmt",
   targetNamespace = "http://org.jboss.ws/samples/retail/profile",
   serviceName = "ProfileMgmtService")
@SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.BARE)      (3)
public class ProfileMgmtBean {

   @WebMethod                                                       (4)
   public DiscountResponse getCustomerDiscount(DiscountRequest request) {
      return new DiscountResponse(request.getCustomer(), 10.00);
   }
}
```

```
1. We are using a stateless session bean implementation
2. Exposed a web service with an explicit namespace
3. It's a doc/lit bare endpoint
4. And offers an 'getCustomerDiscount' operation
```

What about the payload?
The method parameters and return values are going to represent our XML payload and thus require being compatible with JAXB2. Actually you wouldn't need any JAXB annotations for this particular example, because JAXB relies on meaningful defaults. For the sake of documentation we put the more important ones here.

Take a look at the request parameter:

```
package org.jboss.test.ws.jaxws.samples.retail.profile;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlType;

import org.jboss.test.ws.jaxws.samples.retail.Customer;

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(                                                    (1)
  name = "discountRequest",
  namespace="http://org.jboss.ws/samples/retail/profile",
  propOrder = { "customer" }
)
public class DiscountRequest {

   protected Customer customer;

   public DiscountRequest() {
   }

   public DiscountRequest(Customer customer) {
      this.customer = customer;
   }

   public Customer getCustomer() {
      return customer;
   }

   public void setCustomer(Customer value) {
      this.customer = value;
   }

}
```

```
1. In this case we use @XmlType to specify an XML complex type name and override the namespace.
```

 If you have more complex mapping problems you need to consult the JAXB documentation.

Deploying service implementations

Service deployment basically depends on the implementation type. As you may already know web services can be implemented as EJB3 components or plain old Java objects. This quick start leverages EJB3 components, that's why we are going to look at this case in the next sections.

EJB3 services
Simply wrap up the service implementation class, the endpoint interface and any custom data types in a JAR and drop them in the *deployment* directory. No additional deployment descriptors required. Any meta data required for the deployment of the actual web service is taken from the annotations provided on the implementation class and the service endpoint interface. JBossWS will intercept that EJB3 deployment (the bean will also be there) and create an HTTP endpoint at deploy-time.
The JAR package structure
jar -tf jaxws-samples-retail.jar

org/jboss/test/ws/jaxws/samples/retail/profile/DiscountRequest.class
org/jboss/test/ws/jaxws/samples/retail/profile/DiscountResponse.class
org/jboss/test/ws/jaxws/samples/retail/profile/ObjectFactory.class
org/jboss/test/ws/jaxws/samples/retail/profile/ProfileMgmt.class
org/jboss/test/ws/jaxws/samples/retail/profile/ProfileMgmtBean.class
org/jboss/test/ws/jaxws/samples/retail/profile/ProfileMgmtService.class
org/jboss/test/ws/jaxws/samples/retail/profile/package-info.class


If the deployment was successful you should be able to see your endpoint in the application server management console.

Consuming web services

When creating web service clients you would usually start from the WSDL. JBossWS ships with a set of tools to generate the required JAX-WS artefacts to build client implementations. In the following section we will look at the most basic usage patterns. For a more detailed introduction to web service client please consult the user guide.

Creating the client artifacts

Using wsconsume

The *wsconsume* tool is used to consume the abstract contract (WSDL) and produce annotated Java classes (and optionally sources) that define it. We are going to start with the WSDL from our retail example (ProfileMgmtService.wsdl). For a detailed tool reference you need to consult the user guide.

```
wsconsume is a command line tool that generates
portable JAX-WS artifacts from a WSDL file.

usage: org.jboss.ws.tools.jaxws.command.wsconsume [options] <wsdl-url>

options:
    -h, --help                  Show this help message
    -b, --binding=<file>        One or more JAX-WS or JAXB binding files
    -k, --keep                  Keep/Generate Java source
    -c  --catalog=<file>        Oasis XML Catalog file for entity resolution
    -p  --package=<name>        The target package for generated source
    -w  --wsdlLocation=<loc>    Value to use for @WebService.wsdlLocation
    -o, --output=<directory>    The directory to put generated artifacts
    -s, --source=<directory>    The directory to put Java source
    -q, --quiet                 Be somewhat more quiet
    -t, --show-traces           Show full exception stack traces
```

Let's try it on our sample:

```
~./wsconsume.sh -k -p org.jboss.test.ws.jaxws.samples.retail.profile ProfileMgmtService.wsdl
(1)

org/jboss/test/ws/jaxws/samples/retail/profile/Customer.java
org/jboss/test/ws/jaxws/samples/retail/profile/DiscountRequest.java
org/jboss/test/ws/jaxws/samples/retail/profile/DiscountResponse.java
org/jboss/test/ws/jaxws/samples/retail/profile/ObjectFactory.java
org/jboss/test/ws/jaxws/samples/retail/profile/ProfileMgmt.java
org/jboss/test/ws/jaxws/samples/retail/profile/ProfileMgmtService.java
org/jboss/test/ws/jaxws/samples/retail/profile/package-info.java
```

1. As you can see we did use the `-p` switch to specify the package name of the generated sources.

The generated artifacts explained

File	Purpose
ProfileMgmt.java	Service Endpoint Interface
Customer.java	Custom data type
Discount*.java	Custom data type
ObjectFactory.java	JAXB XML Registry
package-info.java	Holder for JAXB package annotations
ProfileMgmtService.java	Service factory

Basically *wsconsume* generates all custom data types (JAXB annotated classes), the service endpoint interface and a service factory class. We will look at how these artifacts can be used the build web service client implementations in the next section.

Constructing a service stub

Web service clients make use of a service stubs that hide the details of a remote web service invocation. To a client application a WS invocation just looks like an invocation of any other business component. In this case the service endpoint interface acts as the business interface. JAX-WS does use a service factory class to construct this as particular service stub:

```
import javax.xml.ws.Service;
[...]
Service service = Service.create(                                   (1)
new URL("http://example.org/service?wsdl"),
new QName("MyService")
);
ProfileMgmt profileMgmt = service.getPort(ProfileMgmt.class);     (2)

// do something with the service stub here...                       (3)
```

1. Create a service factory using the WSDL location and the service name
2. Use the tool created service endpoint interface to build the service stub
3. Use the stub like any other business interface

Appendix

Sample wsdl contract

```
<definitions
    name='ProfileMgmtService'
    targetNamespace='http://org.jboss.ws/samples/retail/profile'
    xmlns='http://schemas.xmlsoap.org/wsdl/'
    xmlns:ns1='http://org.jboss.ws/samples/retail'
    xmlns:soap='http://schemas.xmlsoap.org/wsdl/soap/'
    xmlns:tns='http://org.jboss.ws/samples/retail/profile'
    xmlns:xsd='http://www.w3.org/2001/XMLSchema'>
```

```
<types>

   <xs:schema targetNamespace='http://org.jboss.ws/samples/retail'
              version='1.0' xmlns:xs='http://www.w3.org/2001/XMLSchema'>
      <xs:complexType name='customer'>
         <xs:sequence>
            <xs:element minOccurs='0' name='creditCardDetails' type='xs:string'/>
            <xs:element minOccurs='0' name='firstName' type='xs:string'/>
            <xs:element minOccurs='0' name='lastName' type='xs:string'/>
         </xs:sequence>
      </xs:complexType>
   </xs:schema>

   <xs:schema
       targetNamespace='http://org.jboss.ws/samples/retail/profile'
       version='1.0'
       xmlns:ns1='http://org.jboss.ws/samples/retail'
       xmlns:tns='http://org.jboss.ws/samples/retail/profile'
       xmlns:xs='http://www.w3.org/2001/XMLSchema'>

      <xs:import namespace='http://org.jboss.ws/samples/retail'/>
      <xs:element name='getCustomerDiscount'
                  nillable='true' type='tns:discountRequest'/>
      <xs:element name='getCustomerDiscountResponse'
                  nillable='true' type='tns:discountResponse'/>
      <xs:complexType name='discountRequest'>
         <xs:sequence>
            <xs:element minOccurs='0' name='customer' type='ns1:customer'/>

         </xs:sequence>
      </xs:complexType>
      <xs:complexType name='discountResponse'>
         <xs:sequence>
            <xs:element minOccurs='0' name='customer' type='ns1:customer'/>
            <xs:element name='discount' type='xs:double'/>
         </xs:sequence>
      </xs:complexType>
   </xs:schema>

</types>

<message name='ProfileMgmt_getCustomerDiscount'>
   <part element='tns:getCustomerDiscount' name='getCustomerDiscount'/>
</message>
<message name='ProfileMgmt_getCustomerDiscountResponse'>
   <part element='tns:getCustomerDiscountResponse'
         name='getCustomerDiscountResponse'/>
</message>
<portType name='ProfileMgmt'>
   <operation name='getCustomerDiscount'
              parameterOrder='getCustomerDiscount'>

      <input message='tns:ProfileMgmt_getCustomerDiscount'/>
      <output message='tns:ProfileMgmt_getCustomerDiscountResponse'/>
   </operation>
</portType>
<binding name='ProfileMgmtBinding' type='tns:ProfileMgmt'>
   <soap:binding style='document'
```

```
                    transport='http://schemas.xmlsoap.org/soap/http'/>
      <operation name='getCustomerDiscount'>
         <soap:operation soapAction=''/>
         <input>

            <soap:body use='literal'/>
         </input>
         <output>
            <soap:body use='literal'/>
         </output>
      </operation>
   </binding>
   <service name='ProfileMgmtService'>
      <port binding='tns:ProfileMgmtBinding' name='ProfileMgmtPort'>

         <soap:address
             location='http://<HOST>:<PORT>/jaxws-samples-retail/ProfileMgmtBean'/>
      </port>
   </service>
</definitions>
```

9.2.4 JBoss AS7 Extension Technologies

Coming Soon

This guide is still under development, check back soon!

1. OSGi Technology
2. Management Interface

Management Interface

Coming Soon

This guide is still under development, check back soon!

- Management via the Java Management Extension (JMX)
- Management via RESTful services
- Batch Management / Command Line Interface (CLI)

Management via the Java Management Extension (JMX)

Management via RESTful services

Batch Management / Command Line Interface (CLI)

10 Glossary

10.1 Module

A logical grouping of classes used for classloading and dependency management in WildFly 10. Modules can be *dynamic* or *static*.

Static Modules are the predefined modules installed in the modules/ directory of the application server.

Dynamic Modules are created by the application server for each deployment (or sub-deployment in an EAR).

Reference: Class Loading in WildFly

10.2 Module

A logical grouping of classes used for classloading and dependency management in WildFly 10. Modules can be *dynamic* or *static*.

Static Modules are the predefined modules installed in the modules/ directory of the application server.

Dynamic Modules are created by the application server for each deployment (or sub-deployment in an EAR).

Reference: Class Loading in WildFly

11 Extending WildFly

11.1 Target Audience

This document is intended for people who want to extend WildFly to introduce new capabilities.

11.1.1 Prerequisites

You should know how to download, install and run WildFly. If not please consult the Getting Started Guide. You should also be familiar with the management concepts from the Admin Guide, particularly the Core management concepts section and you need Java development experience to follow the example in this guide.

11.1.2 Examples in this guide

Most of the examples in this guide are being expressed as excerpts of the XML configuration files or by using a representation of the de-typed management model.

11.2 Overview

In this document we provide an example of how to extend the kernel functionality of WildFly via an extension and the subsystem it installs. The WildFly kernel is very simple and lightweight; most of the capabilities people associate with an application server are provided via extensions and their subsystems. The WildFly distribution includes many extensions and subsystems; the webserver integration is via a subsystem; the transaction manager integration is via a subsystem, the EJB container integration is via a subsystem, etc.

This document is divided into two main sections. The first is focused on learning by doing. This section will walk you through the steps needed to create your own subsystem, and will touch on most of the concepts discussed elsewhere in this guide. The second focuses on a conceptual overview of the key interfaces and classes described in the example. Readers should feel free to start with the second section if that better fits their learning style. Jumping back and forth between the sections is also a good strategy.

11.3 Example subsystem

Our example subsystem will keep track of all deployments of certain types containing a special marker file, and expose operations to see how long these deployments have been deployed.

11.3.1 Create the skeleton project

To make your life easier we have provided a maven archetype which will create a skeleton project for implementing subsystems.

```
mvn archetype:generate \
    -DarchetypeArtifactId=wildfly-subsystem \
    -DarchetypeGroupId=org.wildfly.archetypes \
    -DarchetypeVersion=8.0.0.Final \
    -DarchetypeRepository=http://repository.jboss.org/nexus/content/groups/public
```

Maven will download the archetype and it's dependencies, and ask you some questions:

```
$ mvn archetype:generate \
    -DarchetypeArtifactId=wildfly-subsystem \
    -DarchetypeGroupId=org.wildfly.archetypes \
    -DarchetypeVersion=8.0.0.Final \
    -DarchetypeRepository=http://repository.jboss.org/nexus/content/groups/public
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO] ------------------------------------------------------------------------
[INFO]

.........

Define value for property 'groupId': : com.acme.corp
Define value for property 'artifactId': : acme-subsystem
Define value for property 'version':  1.0-SNAPSHOT: :
Define value for property 'package':  com.acme.corp: : com.acme.corp.tracker
Define value for property 'module': : com.acme.corp.tracker
[INFO] Using property: name = WildFly subsystem project
Confirm properties configuration:
groupId: com.acme.corp
artifactId: acme-subsystem
version: 1.0-SNAPSHOT
package: com.acme.corp.tracker
module: com.acme.corp.tracker
name: WildFly subsystem project
 Y: : Y
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1:42.563s
[INFO] Finished at: Fri Jul 08 14:30:09 BST 2011
[INFO] Final Memory: 7M/81M
[INFO] ------------------------------------------------------------------------
$
```

	Instruction
1	Enter the `groupId` you wish to use
2	Enter the `artifactId` you wish to use
3	Enter the version you wish to use, or just hit `Enter` if you wish to accept the default `1.0-SNAPSHOT`
4	Enter the java package you wish to use, or just hit `Enter` if you wish to accept the default (which is copied from `groupId`).
5	Enter the module name you wish to use for your extension.
6	Finally, if you are happy with your choices, hit `Enter` and Maven will generate the project for you.

You can also do this in Eclipse, see Creating your own application for more details. We now have a skeleton project that you can use to implement a subsystem. Import the `acme-subsystem` project into your favourite IDE. A nice side-effect of running this in the IDE is that you can see the javadoc of WildFly classes and interfaces imported by the skeleton code. If you do a `mvn install` in the project it will work if we plug it into WildFly, but before doing that we will change it to do something more useful.

The rest of this section modifies the skeleton project created by the archetype to do something more useful, and the full code can be found in acme-subsystem.zip.

If you do a `mvn install` in the created project, you will see some tests being run

```
$mvn install
[INFO] Scanning for projects...
[...]
[INFO] Surefire report directory:
/Users/kabir/sourcecontrol/temp/archetype-test/acme-subsystem/target/surefire-reports

-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running com.acme.corp.tracker.extension.SubsystemBaseParsingTestCase
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.424 sec
Running com.acme.corp.tracker.extension.SubsystemParsingTestCase
Tests run: 6, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.074 sec

Results :

Tests run: 3, Failures: 0, Errors: 0, Skipped: 0
[...]
```

We will talk about these later in the Testing the parsers section.

11.3.2 Create the schema

First, let us define the schema for our subsystem. Rename `src/main/resources/schema/mysubsystem.xsd` to `src/main/resources/schema/acme.xsd`. Then open `acme.xsd` and modify it to the following

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
            targetNamespace="urn:com.acme.corp.tracker:1.0"
            xmlns="urn:com.acme.corp.tracker:1.0"
            elementFormDefault="qualified"
            attributeFormDefault="unqualified"
            version="1.0">

   <!-- The subsystem root element -->
   <xs:element name="subsystem" type="subsystemType"/>
   <xs:complexType name="subsystemType">
      <xs:all>
         <xs:element name="deployment-types" type="deployment-typesType"/>
      </xs:all>
   </xs:complexType>
   <xs:complexType name="deployment-typesType">
      <xs:choice minOccurs="0" maxOccurs="unbounded">
         <xs:element name="deployment-type" type="deployment-typeType"/>
      </xs:choice>
   </xs:complexType>
   <xs:complexType name="deployment-typeType">
      <xs:attribute name="suffix" use="required"/>
      <xs:attribute name="tick" type="xs:long" use="optional" default="10000"/>
   </xs:complexType>
</xs:schema>
```

Note that we modified the `xmlns` and `targetNamespace` values to `urn.com.acme.corp.tracker:1.0`. Our new `subsystem` element has a child called `deployment-types`, which in turn can have zero or more children called `deployment-type`. Each `deployment-type` has a required `suffix` attribute, and a `tick` attribute which defaults to `true`.
Now modify the `com.acme.corp.tracker.extension.SubsystemExtension` class to contain the new namespace.

```
public class SubsystemExtension implements Extension {

    /** The name space used for the {@code substystem} element */
    public static final String NAMESPACE = "urn:com.acme.corp.tracker:1.0";
    ...
```

11.3.3 Design and define the model structure

The following example xml contains a valid subsystem configuration, we will see how to plug this in to WildFly later in this tutorial.

```
<subsystem xmlns="urn:com.acme.corp.tracker:1.0">
   <deployment-types>
      <deployment-type suffix="sar" tick="10000"/>
      <deployment-type suffix="war" tick="10000"/>
   </deployment-types>
</subsystem>
```

Now when designing our model, we can either do a one to one mapping between the schema and the model or come up with something slightly or very different. To keep things simple, let us stay pretty true to the schema so that when executing a `:read-resource(recursive=true)` against our subsystem we'll see something like:

```
{
    "outcome" => "success",
    "result" => {"type" => {
        "sar" => {"tick" => "10000"},
        "war" => {"tick" => "10000"}
    }}
}
```

Each `deployment-type` in the xml becomes in the model a child resource of the subsystem's root resource. The child resource's child-type is `type`, and it is indexed by its `suffix`. Each `type` resource then contains the `tick` attribute.
We also need a name for our subsystem, to do that change `com.acme.corp.tracker.extension.SubsystemExtension`:

```
public class SubsystemExtension implements Extension {
    ...
    /** The name of our subsystem within the model. */
    public static final String SUBSYSTEM_NAME = "tracker";
    ...
```

Once we are finished our subsystem will be available under `/subsystem=tracker`.
The `SubsystemExtension.initialize()` method defines the model, currently it sets up the basics to add our subsystem to the model:

```
@Override
    public void initialize(ExtensionContext context) {
        //register subsystem with its model version
        final SubsystemRegistration subsystem = context.registerSubsystem(SUBSYSTEM_NAME, 1, 0);
        //register subsystem model with subsystem definition that defines all attributes and
operations
        final ManagementResourceRegistration registration =
subsystem.registerSubsystemModel(SubsystemDefinition.INSTANCE);
        //register describe operation, note that this can be also registered in
SubsystemDefinition
        registration.registerOperationHandler(DESCRIBE,
GenericSubsystemDescribeHandler.INSTANCE, GenericSubsystemDescribeHandler.INSTANCE, false,
OperationEntry.EntryType.PRIVATE);
        //we can register additional submodels here
        //
        subsystem.registerXMLElementWriter(parser);
    }
```

The `registerSubsystem()` call registers our subsystem with the extension context. At the end of the method we register our parser with the returned `SubsystemRegistration` to be able to marshal our subsystem's model back to the main configuration file when it is modified. We will add more functionality to this method later.

Registering the core subsystem model

Next we obtain a `ManagementResourceRegistration` by registering the subsystem model. This is a **compulsory** step for every new subsystem.

```
final ManagementResourceRegistration registration =
subsystem.registerSubsystemModel(SubsystemDefinition.INSTANCE);
```

Its parameter is an implementation of the `ResourceDefinition` interface, which means that when you call `/subsystem=tracker:read-resource-description` the information you see comes from model that is defined by `SubsystemDefinition.INSTANCE`.

```
public class SubsystemDefinition extends SimpleResourceDefinition {
    public static final SubsystemDefinition INSTANCE = new SubsystemDefinition();

    private SubsystemDefinition() {
        super(SubsystemExtension.SUBSYSTEM_PATH,
                SubsystemExtension.getResourceDescriptionResolver(null),
                //We always need to add an 'add' operation
                SubsystemAdd.INSTANCE,
                //Every resource that is added, normally needs a remove operation
                SubsystemRemove.INSTANCE);
    }

    @Override
    public void registerOperations(ManagementResourceRegistration resourceRegistration) {
        super.registerOperations(resourceRegistration);
        //you can register aditional operations here
    }

    @Override
    public void registerAttributes(ManagementResourceRegistration resourceRegistration) {
        //you can register attributes here
    }
}
```

Since we need child resource `type` we need to add new ResourceDefinition,

The `ManagementResourceRegistration` obtained in `SubsystemExtension.initialize()` is then used to add additional operations or to register submodels to the `/subsystem=tracker` address. Every subsystem and resource **must** have an `ADD` method which can be achieved by the following line inside `registerOperations` in your `ResourceDefinition` or by providing it in constructor of your `SimpleResourceDefinition` just as we did in example above.

```
//We always need to add an 'add' operation
        resourceRegistration.registerOperationHandler(ADD, SubsystemAdd.INSTANCE, new
DefaultResourceAddDescriptionProvider(resourceRegistration,descriptionResolver), false);
```

The parameters when registering an operation handler are:

1. **The name** - i.e. `ADD`.
2. The handler instance - we will talk more about this below
3. The handler description provider - we will talk more about this below.
4. Whether this operation handler is inherited - `false` means that this operation is not inherited, and will only apply to `/subsystem=tracker`. The content for this operation handler will be provided by 3.

Let us first look at the description provider which is quite simple since this operation takes no parameters. The addition of `type` children will be handled by another operation handler, as we will see later on.

There are two way to define `DescriptionProvider`, one is by defining it by hand using ModelNode, but as this has show to be very error prone there are lots of helper methods to help you automatically describe the model. Following example is done by manually defining Description provider for ADD operation handler

```
/**
     * Used to create the description of the subsystem add method
     */
    public static DescriptionProvider SUBSYSTEM_ADD = new DescriptionProvider() {
        public ModelNode getModelDescription(Locale locale) {
            //The locale is passed in so you can internationalize the strings used in the
descriptions

            final ModelNode subsystem = new ModelNode();
            subsystem.get(OPERATION_NAME).set(ADD);
            subsystem.get(DESCRIPTION).set("Adds the tracker subsystem");

            return subsystem;
        }
    };
```

Or you can use API that helps you do that for you. For Add and Remove methods there are classes `DefaultResourceAddDescriptionProvider` and `DefaultResourceRemoveDescriptionProvider` that do work for you. In case you use `SimpleResourceDefinition` even that part is hidden from you.

```
resourceRegistration.registerOperationHandler(ADD, SubsystemAdd.INSTANCE, new
DefaultResourceAddDescriptionProvider(resourceRegistration,descriptionResolver), false);
resourceRegistration.registerOperationHandler(REMOVE, SubsystemRemove.INSTANCE, new
DefaultResourceRemoveDescriptionProvider(resourceRegistration,descriptionResolver), false);
```

For other operation handlers that are not add/remove you can use `DefaultOperationDescriptionProvider` that takes additional parameter of what is the name of operation and optional array of parameters/attributes operation takes. This is an example to register operation "`add-mime`" with two parameters:

```
container.registerOperationHandler("add-mime",
                MimeMappingAdd.INSTANCE,
                new DefaultOperationDescriptionProvider("add-mime",
Extension.getResourceDescriptionResolver("container.mime-mapping"), MIME_NAME, MIME_VALUE));
```

When descriping an operation its description provider's `OPERATION_NAME` must match the name used when calling `ManagementResourceRegistration.registerOperationHandler()`

Next we have the actual operation handler instance, note that we have changed its `populateModel()` method to initialize the `type` child of the model.

```
class SubsystemAdd extends AbstractBoottimeAddStepHandler {

    static final SubsystemAdd INSTANCE = new SubsystemAdd();

    private SubsystemAdd() {
    }

    /** {@inheritDoc} */
    @Override
    protected void populateModel(ModelNode operation, ModelNode model) throws
OperationFailedException {
        log.info("Populating the model");
        //Initialize the 'type' child node
        model.get("type").setEmptyObject();
    }
    ....
```

`SubsystemAdd` also has a `performBoottime()` method which is used for initializing the deployer chain associated with this subsystem. We will talk about the deployers later on. However, the basic idea for all operation handlers is that we do any model updates before changing the actual runtime state.

The rule of thumb is that every thing that can be added, can also be removed so we have a remove handler for the subsystem registered
in `SubsystemDefinition.registerOperations` or just provide the operation handler in constructor.

```
//Every resource that is added, normally needs a remove operation
        registration.registerOperationHandler(REMOVE, SubsystemRemove.INSTANCE,
DefaultResourceRemoveDescriptionProvider(resourceRegistration,descriptionResolver) , false);
```

`SubsystemRemove` extends `AbstractRemoveStepHandler` which takes care of removing the resource from the model so we don't need to override its `performRemove()` operation, also the add handler did not install any services (services will be discussed later) so we can delete the `performRuntime()` method generated by the archetype.

```
class SubsystemRemove extends AbstractRemoveStepHandler {

    static final SubsystemRemove INSTANCE = new SubsystemRemove();

    private final Logger log = Logger.getLogger(SubsystemRemove.class);

    private SubsystemRemove() {
    }
}
```

The description provider for the remove operation is simple and quite similar to that of the add handler where just name of the method changes.

Registering the subsystem child

The `type` child does not exist in our skeleton project so we need to implement the operations to add and remove them from the model.

First we need an add operation to add the `type` child, create a class called `com.acme.corp.tracker.extension.TypeAddHandler`. In this case we extend the `org.jboss.as.controller.AbstractAddStepHandler` class and implement the `org.jboss.as.controller.descriptions.DescriptionProvider` interface. `org.jboss.as.controller.OperationStepHandler` is the main interface for the operation handlers, and `AbstractAddStepHandler` is an implementation of that which does the plumbing work for adding a resource to the model.

```
class TypeAddHandler extends AbstractAddStepHandler implements DescriptionProvider {

    public static final TypeAddHandler INSTANCE = new TypeAddHandler();

    private TypeAddHandler() {
    }
```

Then we define subsystem model. Lets call it `TypeDefinition` and for ease of use let it extend `SimpleResourceDefinition` instead just implement `ResourceDefinition`.

```
public class TypeDefinition extends SimpleResourceDefinition {

 public static final TypeDefinition INSTANCE = new TypeDefinition();

 //we define attribute named tick
protected static final SimpleAttributeDefinition TICK =
new SimpleAttributeDefinitionBuilder(TrackerExtension.TICK, ModelType.LONG)
  .setAllowExpression(true)
  .setXmlName(TrackerExtension.TICK)
  .setFlags(AttributeAccess.Flag.RESTART_ALL_SERVICES)
  .setDefaultValue(new ModelNode(1000))
  .setAllowNull(false)
  .build();

private TypeDefinition(){
   super(TYPE_PATH,
TrackerExtension.getResourceDescriptionResolver(TYPE),TypeAdd.INSTANCE,TypeRemove.INSTANCE);
}

@Override
public void registerAttributes(ManagementResourceRegistration resourceRegistration){
   resourceRegistration.registerReadWriteAttribute(TICK, null, TrackerTickHandler.INSTANCE);
}

}
```

Which will take care of describing the model for us. As you can see in example above we define `SimpleAttributeDefinition` named `TICK`, this is a mechanism to define Attributes in more type safe way and to add more common API to manipulate attributes. As you can see here we define default value of 1000 as also other constraints and capabilities. There could be other properties set such as validators, alternate names, xml name, flags for marking it attribute allows expressions and more.
Then we do the work of updating the model by implementing the `populateModel()` method from the `AbstractAddStepHandler`, which populates the model's attribute from the operation parameters. First we get hold of the model relative to the address of this operation (we will see later that we will register it against `/subsystem=tracker/type=*`), so we just specify an empty relative address, and we then populate our model with the parameters from the operation. There is operation `validateAndSet` on `AttributeDefinition` that helps us validate and set the model based on definition of the attribute.

```
@Override
    protected void populateModel(ModelNode operation, ModelNode model) throws
OperationFailedException {
         TICK.validateAndSet(operation,model);
    }
```

We then override the `performRuntime()` method to perform our runtime changes, which in this case involves installing a service into the controller at the heart of WildFly. (`AbstractAddStepHandler.performRuntime()` is similar to `AbstractBoottimeAddStepHandler.performBoottime()` in that the model is updated before runtime changes are made.

```
@Override
    protected void performRuntime(OperationContext context, ModelNode operation, ModelNode
model,
            ServiceVerificationHandler verificationHandler, List<ServiceController<?>>
newControllers)
            throws OperationFailedException {
        String suffix =
PathAddress.pathAddress(operation.get(ModelDescriptionConstants.ADDRESS)).getLastElement().getValu
long tick = TICK.resolveModelAttribute(context,model).asLong();
        TrackerService service = new TrackerService(suffix, tick);
        ServiceName name = TrackerService.createServiceName(suffix);
        ServiceController<TrackerService> controller = context.getServiceTarget()
                .addService(name, service)
                .addListener(verificationHandler)
                .setInitialMode(Mode.ACTIVE)
                .install();
        newControllers.add(controller);
    }
}
```

Since the add methods will be of the format `/subsystem=tracker/suffix=war:add(tick=1234)`, we look for the last element of the operation address, which is `war` in the example just given and use that as our suffix. We then create an instance of TrackerService and install that into the `service target` of the context and add the created `service controller` to the `newControllers` list.

The tracker service is quite simple. All services installed into WildFly must implement the `org.jboss.msc.service.Service` interface.

```
public class TrackerService implements Service<TrackerService>{
```

We then have some fields to keep the tick count and a thread which when run outputs all the deployments registered with our service.

```
private AtomicLong tick = new AtomicLong(10000);

    private Set<String> deployments = Collections.synchronizedSet(new HashSet<String>());
    private Set<String> coolDeployments = Collections.synchronizedSet(new HashSet<String>());
    private final String suffix;

    private Thread OUTPUT = new Thread() {
        @Override
        public void run() {
            while (true) {
                try {
                    Thread.sleep(tick.get());
                    System.out.println("Current deployments deployed while " + suffix + "
tracking active:\n" + deployments
                       + "\nCool: " + coolDeployments.size());
                } catch (InterruptedException e) {
                    interrupted();
                    break;
                }
            }
        }
    };

    public TrackerService(String suffix, long tick) {
        this.suffix = suffix;
        this.tick.set(tick);
    }
```

Next we have three methods which come from the `Service` interface. `getValue()` returns this service, `start()` is called when the service is started by the controller, `stop` is called when the service is stopped by the controller, and they start and stop the thread outputting the deployments.

```
@Override
    public TrackerService getValue() throws IllegalStateException, IllegalArgumentException {
        return this;
    }

    @Override
    public void start(StartContext context) throws StartException {
        OUTPUT.start();
    }

    @Override
    public void stop(StopContext context) {
        OUTPUT.interrupt();
    }
```

Next we have a utility method to create the `ServiceName` which is used to register the service in the controller.

```
public static ServiceName createServiceName(String suffix) {
        return ServiceName.JBOSS.append("tracker", suffix);
}
```

Finally we have some methods to add and remove deployments, and to set and read the `tick`. The 'cool' deployments will be explained later.

```
public void addDeployment(String name) {
        deployments.add(name);
    }

    public void addCoolDeployment(String name) {
        coolDeployments.add(name);
    }

    public void removeDeployment(String name) {
        deployments.remove(name);
        coolDeployments.remove(name);
    }

    void setTick(long tick) {
        this.tick.set(tick);
    }

    public long getTick() {
        return this.tick.get();
    }
}//TrackerService - end
```

Since we are able to add `type` children, we need a way to be able to remove them, so we create a `com.acme.corp.tracker.extension.TypeRemoveHandler`. In this case we extend `AbstractRemoveStepHandler` which takes care of removing the resource from the model so we don't need to override its `performRemove()` operationa. But we need to implement the `DescriptionProvider` method to provide the model description, and since the add handler installs the TrackerService, we need to remove that in the `performRuntime()` method.

```
public class TypeRemoveHandler extends AbstractRemoveStepHandler {

    public static final TypeRemoveHandler INSTANCE = new TypeRemoveHandler();

    private TypeRemoveHandler() {
    }


    @Override
    protected void performRuntime(OperationContext context, ModelNode operation, ModelNode
model) throws OperationFailedException {
        String suffix =
PathAddress.pathAddress(operation.get(ModelDescriptionConstants.ADDRESS)).getLastElement().getValu
ServiceName name = TrackerService.createServiceName(suffix);
        context.removeService(name);
    }

}
```

We then need a description provider for the `type` part of the model itself, so we modify TypeDefinitnion to registerAttribute

```
class TypeDefinition{
...
@Override
public void registerAttributes(ManagementResourceRegistration resourceRegistration){
    resourceRegistration.registerReadWriteAttribute(TICK, null, TrackerTickHandler.INSTANCE);
}

}
```

Then finally we need to specify that our new `type` child and associated handlers go under `/subsystem=tracker/type=*` in the model by adding registering it with the model in `SubsystemExtension.initialize()`. So we add the following just before the end of the method.

```
@Override
public void initialize(ExtensionContext context)
{
 final SubsystemRegistration subsystem = context.registerSubsystem(SUBSYSTEM_NAME, 1, 0);
 final ManagementResourceRegistration registration =
subsystem.registerSubsystemModel(TrackerSubsystemDefinition.INSTANCE);
 //Add the type child
 ManagementResourceRegistration typeChild =
registration.registerSubModel(TypeDefinition.INSTANCE);
 subsystem.registerXMLElementWriter(parser);
}
```

The above first creates a child of our main subsystem registration for the relative address `type=*`, and gets the `typeChild` registration.
To this we add the `TypeAddHandler` and `TypeRemoveHandler`.
The add variety is added under the name `add` and the remove handler under the name `remove`, and for each registered operation handler we use the handler singleton instance as both the handler parameter and as the `DescriptionProvider`.

Finally, we register `tick` as a read/write attribute, the null parameter means we don't do anything special with regards to reading it, for the write handler we supply it with an operation handler called `TrackerTickHandler`.
Registering it as a read/write attribute means we can use the `:write-attribute` operation to modify the value of the parameter, and it will be handled by `TrackerTickHandler`.

Not registering a write attribute handler makes the attribute read only.

`TrackerTickHandler` extends `AbstractWriteAttributeHandler`
directly, and so must implement its `applyUpdateToRuntime` and `revertUpdateToRuntime` method.
This takes care of model manipulation (validation, setting) but leaves us to do just to deal with what we need to do.

```
class TrackerTickHandler extends AbstractWriteAttributeHandler<Void> {

    public static final TrackerTickHandler INSTANCE = new TrackerTickHandler();

    private TrackerTickHandler() {
        super(TypeDefinition.TICK);
    }

    protected boolean applyUpdateToRuntime(OperationContext context, ModelNode operation, String
attributeName,
              ModelNode resolvedValue, ModelNode currentValue, HandbackHolder<Void>
handbackHolder) throws OperationFailedException {

        modifyTick(context, operation, resolvedValue.asLong());

        return false;
    }

    protected void revertUpdateToRuntime(OperationContext context, ModelNode operation, String
attributeName, ModelNode valueToRestore, ModelNode valueToRevert, Void handback){
        modifyTick(context, operation, valueToRestore.asLong());
    }

    private void modifyTick(OperationContext context, ModelNode operation, long value) throws
OperationFailedException {

        final String suffix =
PathAddress.pathAddress(operation.get(ModelDescriptionConstants.ADDRESS)).getLastElement().getValu
TrackerService service = (TrackerService)
context.getServiceRegistry(true).getRequiredService(TrackerService.createServiceName(suffix)).getV
service.setTick(value);
    }

}
```

The operation used to execute this will be of the form `/subsystem=tracker/type=war:write-attribute(name=tick,value=12345)` so we first get the `suffix` from the operation address, and the `tick` value from the operation parameter's `resolvedValue` parameter, and use that to update the model.

We then add a new step associated with the `RUNTIME` stage to update the tick of the TrackerService for our suffix. This is essential since the call to `context.getServiceRegistry()` will fail unless the step accessing it belongs to the `RUNTIME` stage.

When implementing `execute()`, you **must** call `context.completeStep()` when you are done.

11.3.4 Parsing and marshalling of the subsystem xml

WildFly uses the Stax API to parse the xml files. This is initialized in `SubsystemExtension` by mapping our parser onto our namespace:

```
public class SubsystemExtension implements Extension {

    /** The name space used for the {@code subsystem} element */
    public static final String NAMESPACE = "urn:com.acme.corp.tracker:1.0";
    ...
    protected static final PathElement SUBSYSTEM_PATH = PathElement.pathElement(SUBSYSTEM,
SUBSYSTEM_NAME);
    protected static final PathElement TYPE_PATH = PathElement.pathElement(TYPE);

   /** The parser used for parsing our subsystem */
    private final SubsystemParser parser = new SubsystemParser();

   @Override
    public void initializeParsers(ExtensionParsingContext context) {
        context.setSubsystemXmlMapping(NAMESPACE, parser);
    }
    ...
```

We then need to write the parser. The contract is that we read our subsystem's xml and create the operations that will populate the model with the state contained in the xml. These operations will then be executed on our behalf as part of the parsing process. The entry point is the `readElement()` method.

```
public class SubsystemExtension implements Extension {

    /**
     * The subsystem parser, which uses stax to read and write to and from xml
     */
    private static class SubsystemParser implements XMLStreamConstants,
XMLElementReader<List<ModelNode>>, XMLElementWriter<SubsystemMarshallingContext> {

        /** {@inheritDoc} */
        @Override
        public void readElement(XMLExtendedStreamReader reader, List<ModelNode> list) throws
XMLStreamException {
            // Require no attributes
            ParseUtils.requireNoAttributes(reader);

            //Add the main subsystem 'add' operation
            final ModelNode subsystem = new ModelNode();
            subsystem.get(OP).set(ADD);
            subsystem.get(OP_ADDR).set(PathAddress.pathAddress(SUBSYSTEM_PATH).toModelNode());
            list.add(subsystem);

            //Read the children
            while (reader.hasNext() && reader.nextTag() != END_ELEMENT) {
                if (!reader.getLocalName().equals("deployment-types")) {
```

```
                    throw ParseUtils.unexpectedElement(reader);
                }
                while (reader.hasNext() && reader.nextTag() != END_ELEMENT) {
                    if (reader.isStartElement()) {
                        readDeploymentType(reader, list);
                    }
                }
            }
        }

        private void readDeploymentType(XMLExtendedStreamReader reader, List<ModelNode> list)
throws XMLStreamException {
            if (!reader.getLocalName().equals("deployment-type")) {
                throw ParseUtils.unexpectedElement(reader);
            }
            ModelNode addTypeOperation = new ModelNode();
            addTypeOperation.get(OP).set(ModelDescriptionConstants.ADD);

            String suffix = null;
            for (int i = 0; i < reader.getAttributeCount(); i++) {
                String attr = reader.getAttributeLocalName(i);
                String value = reader.getAttributeValue(i);
                if (attr.equals("tick")) {
                    TypeDefinition.TICK.parseAndSetParameter(value, addTypeOperation, reader);
                } else if (attr.equals("suffix")) {
                    suffix = value;
                } else {
                    throw ParseUtils.unexpectedAttribute(reader, i);
                }
            }
            ParseUtils.requireNoContent(reader);
            if (suffix == null) {
                throw ParseUtils.missingRequiredElement(reader,
Collections.singleton("suffix"));
            }

            //Add the 'add' operation for each 'type' child
            PathAddress addr = PathAddress.pathAddress(SUBSYSTEM_PATH,
PathElement.pathElement(TYPE, suffix));
            addTypeOperation.get(OP_ADDR).set(addr.toModelNode());
            list.add(addTypeOperation);
        }
        ...
```

So in the above we always create the add operation for our subsystem. Due to its address `/subsystem=tracker` defined by SUBSYSTEM_PATH this will trigger the `SubsystemAddHandler` we created earlier when we invoke `/subsystem=tracker:add`. We then parse the child elements and create an add operation for the child address for each `type` child. Since the address will for example be `/subsystem=tracker/type=sar` (defined by TYPE_PATH) and `TypeAddHandler` is registered for all `type` subaddresses the `TypeAddHandler` will get invoked for those operations. Note that when we are parsing attribute `tick` we are using definition of attribute that we defined in TypeDefintion to parse attribute value and apply all rules that we specified for this attribute, this also enables us to property support expressions on attributes.

The parser is also used to marshal the model to xml whenever something modifies the model, for which the entry point is the `writeContent()` method:

```
private static class SubsystemParser implements XMLStreamConstants,
XMLElementReader<List<ModelNode>>, XMLElementWriter<SubsystemMarshallingContext> {
        ...
        /** {@inheritDoc} */
        @Override
        public void writeContent(final XMLExtendedStreamWriter writer, final
SubsystemMarshallingContext context) throws XMLStreamException {
            //Write out the main subsystem element
            context.startSubsystemElement(TrackerExtension.NAMESPACE, false);
            writer.writeStartElement("deployment-types");
            ModelNode node = context.getModelNode();
            ModelNode type = node.get(TYPE);
            for (Property property : type.asPropertyList()) {

                //write each child element to xml
                writer.writeStartElement("deployment-type");
                writer.writeAttribute("suffix", property.getName());
                ModelNode entry = property.getValue();
                TypeDefinition.TICK.marshallAsAttribute(entry, true, writer);
                writer.writeEndElement();
            }
            //End deployment-types
            writer.writeEndElement();
            //End subsystem
            writer.writeEndElement();
        }
    }
```

Then we have to implement the `SubsystemDescribeHandler` which translates the current state of the model into operations similar to the ones created by the parser. The `SubsystemDescribeHandler` is only used when running in a managed domain, and is used when the host controller queries the domain controller for the configuration of the profile used to start up each server. In our case the `SubsystemDescribeHandler` adds the operation to add the subsystem and then adds the operation to add each `type` child. Since we are using ResourceDefinitinon for defining subsystem all that is generated for us, but if you want to customize that you can do it by implementing it like this.

```
private static class SubsystemDescribeHandler implements OperationStepHandler,
DescriptionProvider {
        static final SubsystemDescribeHandler INSTANCE = new SubsystemDescribeHandler();

        public void execute(OperationContext context, ModelNode operation) throws
OperationFailedException {
            //Add the main operation
            context.getResult().add(createAddSubsystemOperation());

            //Add the operations to create each child

            ModelNode node = context.readModel(PathAddress.EMPTY_ADDRESS);
            for (Property property : node.get("type").asPropertyList()) {

                ModelNode addType = new ModelNode();
                addType.get(OP).set(ModelDescriptionConstants.ADD);
                PathAddress addr = PathAddress.pathAddress(SUBSYSTEM_PATH,
PathElement.pathElement("type", property.getName()));
                addType.get(OP_ADDR).set(addr.toModelNode());
                if (property.getValue().hasDefined("tick")) {
                   TypeDefinition.TICK.validateAndSet(property,addType);
                }
                context.getResult().add(addType);
            }
            context.completeStep();
        }


}
```

Testing the parsers

Changes to tests between 7.0.0 and 7.0.1

The testing framework was moved from the archetype into the core JBoss AS 7 sources between JBoss AS 7.0.0 and JBoss AS 7.0.1, and has been improved upon and is used internally for testing JBoss AS 7's subsystems. The differences between the two versions is that in 7.0.0.Final the testing framework is bundled with the code generated by the archetype (in a sub-package of the package specified for your subsystem, e.g. `com.acme.corp.tracker.support`), and the test extends the `AbstractParsingTest` class.

From 7.0.1 the testing framework is now brought in via the `org.jboss.as:jboss-as-subsystem-test` maven artifact, and the test's superclass is `org.jboss.as.subsystem.test.AbstractSubsystemTest`. The concepts are the same but more and more functionality will be available as JBoss AS 7 is developed.

Now that we have modified our parsers we need to update our tests to reflect the new model. There are currently three tests testing the basic functionality, something which is a lot easier to debug from your IDE before you plug it into the application server. We will talk about these tests in turn and they all live in `com.acme.corp.tracker.extension.SubsystemParsingTestCase`. `SubsystemParsingTestCase` extends `AbstractSubsystemTest` which does a lot of the setup for you and contains utility methods for verifying things from your test. See the javadoc of that class for more information about the functionality available to you. And by all means feel free to add more tests for your subsystem, here we are only testing for the best case scenario while you will probably want to throw in a few tests for edge cases.

The first test we need to modify is `testParseSubsystem()`. It tests that the parsed xml becomes the expected operations that will be parsed into the server, so let us tweak this test to match our subsystem. First we tell the test to parse the xml into operations

```
@Test
    public void testParseSubsystem() throws Exception {
        //Parse the subsystem xml into operations
        String subsystemXml =
                "<subsystem xmlns=\"" + SubsystemExtension.NAMESPACE + "\">" +
                "   <deployment-types>" +
                "       <deployment-type suffix=\"tst\" tick=\"12345\"/>" +
                "   </deployment-types>" +
                "</subsystem>";
        List<ModelNode> operations = super.parse(subsystemXml);
```

There should be one operation for adding the subsystem itself and an operation for adding the `deployment-type`, so check we got two operations

```
///Check that we have the expected number of operations
        Assert.assertEquals(2, operations.size());
```

Now check that the first operation is `add` for the address `/subsystem=tracker`:

```
//Check that each operation has the correct content
        //The add subsystem operation will happen first
        ModelNode addSubsystem = operations.get(0);
        Assert.assertEquals(ADD, addSubsystem.get(OP).asString());
        PathAddress addr = PathAddress.pathAddress(addSubsystem.get(OP_ADDR));
        Assert.assertEquals(1, addr.size());
        PathElement element = addr.getElement(0);
        Assert.assertEquals(SUBSYSTEM, element.getKey());
        Assert.assertEquals(SubsystemExtension.SUBSYSTEM_NAME, element.getValue());
```

Then check that the second operation is `add` for the address `/subsystem=tracker`, and that `12345` was picked up for the value of the `tick` parameter:

```
//Then we will get the add type operation
        ModelNode addType = operations.get(1);
        Assert.assertEquals(ADD, addType.get(OP).asString());
        Assert.assertEquals(12345, addType.get("tick").asLong());
        addr = PathAddress.pathAddress(addType.get(OP_ADDR));
        Assert.assertEquals(2, addr.size());
        element = addr.getElement(0);
        Assert.assertEquals(SUBSYSTEM, element.getKey());
        Assert.assertEquals(SubsystemExtension.SUBSYSTEM_NAME, element.getValue());
        element = addr.getElement(1);
        Assert.assertEquals("type", element.getKey());
        Assert.assertEquals("tst", element.getValue());
    }
```

The second test we need to modify is `testInstallIntoController()` which tests that the xml installs properly into the controller. In other words we are making sure that the `add` operations we created earlier work properly. First we create the xml and install it into the controller. Behind the scenes this will parse the xml into operations as we saw in the last test, but it will also create a new controller and boot that up using the created operations

```
@Test
    public void testInstallIntoController() throws Exception {
        //Parse the subsystem xml and install into the controller
        String subsystemXml =
                "<subsystem xmlns=\"" + SubsystemExtension.NAMESPACE + "\">" +
                "   <deployment-types>" +
                "       <deployment-type suffix=\"tst\" tick=\"12345\"/>" +
                "   </deployment-types>" +
                "</subsystem>";
        KernelServices services = super.installInController(subsystemXml);
```

The returned `KernelServices` allow us to execute operations on the controller, and to read the whole model.

```
//Read the whole model and make sure it looks as expected
        ModelNode model = services.readWholeModel();
        //Useful for debugging :-)
        //System.out.println(model);
```

Now we make sure that the structure of the model within the controller has the expected format and values

```
Assert.assertTrue(model.get(SUBSYSTEM).hasDefined(SubsystemExtension.SUBSYSTEM_NAME));
        Assert.assertTrue(model.get(SUBSYSTEM,
SubsystemExtension.SUBSYSTEM_NAME).hasDefined("type"));
        Assert.assertTrue(model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME,
"type").hasDefined("tst"));
        Assert.assertTrue(model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME, "type",
"tst").hasDefined("tick"));
        Assert.assertEquals(12345, model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME,
"type", "tst", "tick").asLong());
    }
```

The last test provided is called `testParseAndMarshalModel()`. It's main purpose is to make sure that our `SubsystemParser.writeContent()` works as expected. This is achieved by starting a controller in the same way as before

```
@Test
    public void testParseAndMarshalModel() throws Exception {
        //Parse the subsystem xml and install into the first controller
        String subsystemXml =
                "<subsystem xmlns=\"" + SubsystemExtension.NAMESPACE + "\">" +
                "   <deployment-types>" +
                "       <deployment-type suffix=\"tst\" tick=\"12345\"/>" +
                "   </deployment-types>" +
                "</subsystem>";
        KernelServices servicesA = super.installInController(subsystemXml);
```

Now we read the model and the xml that was persisted from the first controller, and use that xml to start a second controller

```
//Get the model and the persisted xml from the first controller
        ModelNode modelA = servicesA.readWholeModel();
        String marshalled = servicesA.getPersistedSubsystemXml();

        //Install the persisted xml from the first controller into a second controller
        KernelServices servicesB = super.installInController(marshalled);
```

Finally we read the model from the second controller, and make sure that the models are identical by calling `compare()` on the test superclass.

```
ModelNode modelB = servicesB.readWholeModel();

        //Make sure the models from the two controllers are identical
        super.compare(modelA, modelB);
    }
```

We then have a test that needs no changing from what the archetype provides us with. As we have seen before we start a controller

```
@Test
    public void testDescribeHandler() throws Exception {
        //Parse the subsystem xml and install into the first controller
        String subsystemXml =
                "<subsystem xmlns=\"" + SubsystemExtension.NAMESPACE + "\">" +
                "</subsystem>";
        KernelServices servicesA = super.installInController(subsystemXml);
```

We then call `/subsystem=tracker:describe` which outputs the subsystem as operations needed to reach the current state (Done by our `SubsystemDescribeHandler`)

```
//Get the model and the describe operations from the first controller
        ModelNode modelA = servicesA.readWholeModel();
        ModelNode describeOp = new ModelNode();
        describeOp.get(OP).set(DESCRIBE);
        describeOp.get(OP_ADDR).set(
                PathAddress.pathAddress(
                        PathElement.pathElement(SUBSYSTEM,
SubsystemExtension.SUBSYSTEM_NAME)).toModelNode());
        List<ModelNode> operations =
super.checkResultAndGetContents(servicesA.executeOperation(describeOp)).asList();
```

Then we create a new controller using those operations

```
//Install the describe options from the first controller into a second controller
        KernelServices servicesB = super.installInController(operations);
```

And then we read the model from the second controller and make sure that the two subsystems are identical
ModelNode modelB = servicesB.readWholeModel();

```
//Make sure the models from the two controllers are identical
        super.compare(modelA, modelB);

    }
```

To test the removal of the the subsystem and child resources we modify the `testSubsystemRemoval()` test provided by the archetype:

```
/**
     * Tests that the subsystem can be removed
     */
    @Test
    public void testSubsystemRemoval() throws Exception {
        //Parse the subsystem xml and install into the first controller
```

We provide xml for the subsystem installing a child, which in turn installs a TrackerService

```
String subsystemXml =
                "<subsystem xmlns=\"" + SubsystemExtension.NAMESPACE + "\">" +
                "   <deployment-types>" +
                "       <deployment-type suffix=\"tst\" tick=\"12345\"/>" +
                "   </deployment-types>" +
                "</subsystem>";
        KernelServices services = super.installInController(subsystemXml);
```

Having installed the xml into the controller we make sure the TrackerService is there

```
//Sanity check to test the service for 'tst' was there
        services.getContainer().getRequiredService(TrackerService.createServiceName("tst"));
```

This call from the subsystem test harness will call remove for each level in our subsystem, children first and validate
that the subsystem model is empty at the end.

```
//Checks that the subsystem was removed from the model
        super.assertRemoveSubsystemResources(services);
```

Finally we check that all the services were removed by the remove handlers

```
//Check that any services that were installed were removed here
        try {
            services.getContainer().getRequiredService(TrackerService.createServiceName("tst"));
            Assert.fail("Should have removed services");
        } catch (Exception expected) {
        }
    }
```

For good measure let us throw in another test which adds a `deployment-type` and also changes its attribute at runtime. So first of all boot up the controller with the same xml we have been using so far

```
@Test
    public void testExecuteOperations() throws Exception {
        String subsystemXml =
                "<subsystem xmlns=\"" + SubsystemExtension.NAMESPACE + "\">" +
                "   <deployment-types>" +
                "       <deployment-type suffix=\"tst\" tick=\"12345\"/>" +
                "   </deployment-types>" +
                "</subsystem>";
        KernelServices services = super.installInController(subsystemXml);
```

Now create an operation which does the same as the following CLI command
`/subsystem=tracker/type=foo:add(tick=1000)`

```
//Add another type
        PathAddress fooTypeAddr = PathAddress.pathAddress(
                PathElement.pathElement(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME),
                PathElement.pathElement("type", "foo"));
        ModelNode addOp = new ModelNode();
        addOp.get(OP).set(ADD);
        addOp.get(OP_ADDR).set(fooTypeAddr.toModelNode());
        addOp.get("tick").set(1000);
```

Execute the operation and make sure it was successful

```
ModelNode result = services.executeOperation(addOp);
        Assert.assertEquals(SUCCESS, result.get(OUTCOME).asString());
```

Read the whole model and make sure that the original data is still there (i.e. the same as what was done by `testInstallIntoController()`

```
ModelNode model = services.readWholeModel();
        Assert.assertTrue(model.get(SUBSYSTEM).hasDefined(SubsystemExtension.SUBSYSTEM_NAME));
        Assert.assertTrue(model.get(SUBSYSTEM,
SubsystemExtension.SUBSYSTEM_NAME).hasDefined("type"));
        Assert.assertTrue(model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME,
"type").hasDefined("tst"));
        Assert.assertTrue(model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME, "type",
"tst").hasDefined("tick"));
        Assert.assertEquals(12345, model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME,
"type", "tst", "tick").asLong());
```

Then make sure our new `type` has been added:

```
Assert.assertTrue(model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME,
"type").hasDefined("foo"));
        Assert.assertTrue(model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME, "type",
"foo").hasDefined("tick"));
        Assert.assertEquals(1000, model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME,
"type", "foo", "tick").asLong());
```

Then we call `write-attribute` to change the `tick` value of `/subsystem=tracker/type=foo`:

```
//Call write-attribute
        ModelNode writeOp = new ModelNode();
        writeOp.get(OP).set(WRITE_ATTRIBUTE_OPERATION);
        writeOp.get(OP_ADDR).set(fooTypeAddr.toModelNode());
        writeOp.get(NAME).set("tick");
        writeOp.get(VALUE).set(3456);
        result = services.executeOperation(writeOp);
        Assert.assertEquals(SUCCESS, result.get(OUTCOME).asString());
```

To give you exposure to other ways of doing things, now instead of reading the whole model to check the attribute, we call `read-attribute` instead, and make sure it has the value we set it to.

```
//Check that write attribute took effect, this time by calling read-attribute instead of reading
the whole model
        ModelNode readOp = new ModelNode();
        readOp.get(OP).set(READ_ATTRIBUTE_OPERATION);
        readOp.get(OP_ADDR).set(fooTypeAddr.toModelNode());
        readOp.get(NAME).set("tick");
        result = services.executeOperation(readOp);
        Assert.assertEquals(3456, checkResultAndGetContents(result).asLong());
```

Since each `type` installs its own copy of `TrackerService`, we get the `TrackerService` for `type=foo` from the service container exposed by the kernel services and make sure it has the right value

```
TrackerService service =
(TrackerService)services.getContainer().getService(TrackerService.createServiceName("foo")).getVal
Assert.assertEquals(3456, service.getTick());
    }
```

TypeDefinition.TICK.

11.3.5 Add the deployers

When discussing `SubsystemAddHandler` we did not mention the work done to install the deployers, which is done in the following method:

```
@Override
    public void performBoottime(OperationContext context, ModelNode operation, ModelNode model,
            ServiceVerificationHandler verificationHandler, List<ServiceController<?>>
newControllers)
            throws OperationFailedException {

        log.info("Populating the model");

        //Add deployment processors here
        //Remove this if you don't need to hook into the deployers, or you can add as many as
you like
        //see SubDeploymentProcessor for explanation of the phases
        context.addStep(new AbstractDeploymentChainStep() {
            public void execute(DeploymentProcessorTarget processorTarget) {
                processorTarget.addDeploymentProcessor(SubsystemDeploymentProcessor.PHASE,
SubsystemDeploymentProcessor.priority, new SubsystemDeploymentProcessor());

            }
        }, OperationContext.Stage.RUNTIME);

    }
```

This adds an extra step which is responsible for installing deployment processors. You can add as many as you like, or avoid adding any all together depending on your needs. Each processor has a `Phase` and a `priority`. Phases are sequential, and a deployment passes through each phases deployment processors. The `priority` specifies where within a phase the processor appears. See `org.jboss.as.server.deployment.Phase` for more information about phases.

In our case we are keeping it simple and staying with one deployment processor with the phase and priority created for us by the maven archetype. The phases will be explained in the next section. The deployment processor is as follows:

```
public class SubsystemDeploymentProcessor implements DeploymentUnitProcessor {
    ...

    @Override
    public void deploy(DeploymentPhaseContext phaseContext) throws
DeploymentUnitProcessingException {
        String name = phaseContext.getDeploymentUnit().getName();
        TrackerService service = getTrackerService(phaseContext.getServiceRegistry(), name);
        if (service != null) {
            ResourceRoot root =
phaseContext.getDeploymentUnit().getAttachment(Attachments.DEPLOYMENT_ROOT);
            VirtualFile cool = root.getRoot().getChild("META-INF/cool.txt");
            service.addDeployment(name);
            if (cool.exists()) {
                service.addCoolDeployment(name);
            }
        }
    }

    @Override
    public void undeploy(DeploymentUnit context) {
        context.getServiceRegistry();
        String name = context.getName();
        TrackerService service = getTrackerService(context.getServiceRegistry(), name);
        if (service != null) {
            service.removeDeployment(name);
        }
    }

    private TrackerService getTrackerService(ServiceRegistry registry, String name) {
        int last = name.lastIndexOf(".");
        String suffix = name.substring(last + 1);
        ServiceController<?> container =
registry.getService(TrackerService.createServiceName(suffix));
        if (container != null) {
            TrackerService service = (TrackerService)container.getValue();
            return service;
        }
        return null;
    }
}
```

The `deploy()` method is called when a deployment is being deployed. In this case we look for the `TrackerService` instance for the service name created from the deployment's suffix. If there is one it means that we are meant to be tracking deployments with this suffix (i.e. `TypeAddHandler` was called for this suffix), and if we find one we add the deployment's name to it. Similarly `undeploy()` is called when a deployment is being undeployed, and if there is a `TrackerService` instance for the deployment's suffix, we remove the deployment's name from it.

Deployment phases and attachments

The code in the SubsystemDeploymentProcessor uses an *attachment*, which is the means of communication between the individual deployment processors. A deployment processor belonging to a phase may create an attachment which is then read further along the chain of deployment unit processors. In the above example we look for the `Attachments.DEPLOYMENT_ROOT` attachment, which is a view of the file structure of the deployment unit put in place before the chain of deployment unit processors is invoked.

As mentioned above, the deployment unit processors are organized in phases, and have a relative order within each phase. A deployment unit passes through all the deployment unit processors in that order. A deployment unit processor may choose to take action or not depending on what attachments are available. Let's take a quick look at what the deployment unit processors for in the phases described in `org.jboss.as.server.deployment.Phase`.

STRUCTURE

The deployment unit processors in this phase determine the structure of a deployment, and looks for sub deployments and metadata files.

PARSE

In this phase the deployment unit processors parse the deployment descriptors and build up the annotation index. `Class-Path` entries from the META-INF/MANIFEST.MF are added.

DEPENDENCIES

Extra class path dependencies are added. For example if deploying a `war` file, the commonly needed dependencies for a web application are added.

CONFIGURE_MODULE

In this phase the modular class loader for the deployment is created. No attempt should be made loading classes from the deployment until **after** this phase.

POST_MODULE

Now that our class loader has been constructed we have access to the classes. In this stage deployment processors may use the `Attachments.REFLECTION_INDEX` attachment which is a deployment index used to obtain members of classes in the deployment, and to invoke upon them, bypassing the inefficiencies of using `java.lang.reflect` directly.

INSTALL

Install new services coming from the deployment.

CLEANUP

Attachments put in place earlier in the deployment unit processor chain may be removed here.

11.3.6 Integrate with WildFly

Now that we have all the code needed for our subsystem, we can build our project by running `mvn install`

```
[kabir ~/sourcecontrol/temp/archetype-test/acme-subsystem]
$mvn install
[INFO] Scanning for projects...
[...]
main:
   [delete] Deleting:
/Users/kabir/sourcecontrol/temp/archetype-test/acme-subsystem/null1004283288
   [delete] Deleting directory
/Users/kabir/sourcecontrol/temp/archetype-test/acme-subsystem/target/module
     [copy] Copying 1 file to
/Users/kabir/sourcecontrol/temp/archetype-test/acme-subsystem/target/module/com/acme/corp/tracker/
[copy] Copying 1 file to
/Users/kabir/sourcecontrol/temp/archetype-test/acme-subsystem/target/module/com/acme/corp/tracker/
[echo] Module com.acme.corp.tracker has been created in the target/module directory. Copy to
your JBoss AS 7 installation.
[INFO] Executed tasks
[INFO]
[INFO] --- maven-install-plugin:2.3.1:install (default-install) @ acme-subsystem ---
[INFO] Installing
/Users/kabir/sourcecontrol/temp/archetype-test/acme-subsystem/target/acme-subsystem.jar to
/Users/kabir/.m2/repository/com/acme/corp/acme-subsystem/1.0-SNAPSHOT/acme-subsystem-1.0-SNAPSHOT.
Installing /Users/kabir/sourcecontrol/temp/archetype-test/acme-subsystem/pom.xml to
/Users/kabir/.m2/repository/com/acme/corp/acme-subsystem/1.0-SNAPSHOT/acme-subsystem-1.0-SNAPSHOT.
------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 5.851s
[INFO] Finished at: Mon Jul 11 23:24:58 BST 2011
[INFO] Final Memory: 7M/81M
[INFO] ------------------------------------------------------------------------
```

This will have built our project and assembled a module for us that can be used for installing it into WildFly. If you go to the `target/module` folder where you built the project you will see the module

```
$ls target/module/com/acme/corp/tracker/main/
acme-subsystem.jar  module.xml
```

The `module.xml` comes from `src/main/resources/module/main/module.xml` and is used to define your module. It says that it contains the `acme-subsystem.jar`:

```
<module xmlns="urn:jboss:module:1.0" name="com.acme.corp.tracker">
    <resources>
        <resource-root path="acme-subsystem.jar"/>
    </resources>
```

And has a default set of dependencies needed by every subsystem created. If your subsystem requires additional module dependencies you can add them here before building and installing.

```
<dependencies>
        <module name="javax.api"/>
        <module name="org.jboss.staxmapper"/>
        <module name="org.jboss.as.controller"/>
        <module name="org.jboss.as.server"/>
        <module name="org.jboss.modules"/>
        <module name="org.jboss.msc"/>
        <module name="org.jboss.logging"/>
        <module name="org.jboss.vfs"/>
    </dependencies>
</module>
```

Note that the name of the module corresponds to the directory structure containing it. Now copy the `target/module/com/acme/corp/tracker/main/` directory and its contents to `$WFLY/modules/com/acme/corp/tracker/main/` (where `$WFLY` is the root of your WildFly install).

Next we need to modify `$WFLY/standalone/configuration/standalone.xml`. First we need to add our new module to the `<extensions>` section:

```
<extensions>
        ...
        <extension module="org.jboss.as.weld"/>
        <extension module="com.acme.corp.tracker"/>
    </extensions>
```

And then we have to add our subsystem to the `<profile>` section:

```
<profile>
    ...

        <subsystem xmlns="urn:com.acme.corp.tracker:1.0">
            <deployment-types>
                <deployment-type suffix="sar" tick="10000"/>
                <deployment-type suffix="war" tick="10000"/>
            </deployment-types>
        </subsystem>
    ...
    </profile>
```

Adding this to a managed domain works exactly the same apart from in this case you need to modify `$WFLY/domain/configuration/domain.xml`.

Now start up WildFly by running `$WFLY/bin/standalone.sh` and you should see messages like these after the server has started, which means our subsystem has been added and our `TrackerService` is working:

```
15:27:33,838 INFO  [org.jboss.as] (Controller Boot Thread) JBoss AS 7.0.0.Final "Lightning"
started in 2861ms - Started 94 of 149 services (55 services are passive or on-demand)
15:27:42,966 INFO  [stdout] (Thread-8) Current deployments deployed while sar tracking active:
15:27:42,966 INFO  [stdout] (Thread-8) []
15:27:42,967 INFO  [stdout] (Thread-8) Cool: 0
15:27:42,967 INFO  [stdout] (Thread-9) Current deployments deployed while war tracking active:
15:27:42,967 INFO  [stdout] (Thread-9) []
15:27:42,967 INFO  [stdout] (Thread-9) Cool: 0
15:27:52,967 INFO  [stdout] (Thread-8) Current deployments deployed while sar tracking active:
15:27:52,967 INFO  [stdout] (Thread-8) []
15:27:52,967 INFO  [stdout] (Thread-8) Cool: 0
```

If you run the command line interface you can execute some commands to see more about the subsystem. For example

```
[standalone@localhost:9999 /] /subsystem=tracker/:read-resource-description(recursive=true,
operations=true)
```

will return a lot of information, including what we provided in the `DescriptionProviders` we created to document our subsystem.

To see the current subsystem state you can execute

```
[standalone@localhost:9999 /] /subsystem=tracker/:read-resource(recursive=true)
{
    "outcome" => "success",
    "result" => {"type" => {
        "war" => {"tick" => 10000L},
        "sar" => {"tick" => 10000L}
    }}
}
```

We can remove both the deployment types which removes them from the model:

```
[standalone@localhost:9999 /] /subsystem=tracker/type=sar:remove
{"outcome" => "success"}
[standalone@localhost:9999 /] /subsystem=tracker/type=war:remove
{"outcome" => "success"}
[standalone@localhost:9999 /] /subsystem=tracker/:read-resource(recursive=true)
{
    "outcome" => "success",
    "result" => {"type" => undefined}
}
```

You should now see the output from the `TrackerService` instances having stopped.

Now, let's add the war tracker again:

```
[standalone@localhost:9999 /] /subsystem=tracker/type=war:add
{"outcome" => "success"}
[standalone@localhost:9999 /] /subsystem=tracker/:read-resource(recursive=true)
{
    "outcome" => "success",
    "result" => {"type" => {"war" => {"tick" => 10000L}}}
}
```

and the WildFly console should show the messages coming from the war `TrackerService` again.

Now let us deploy something. You can find two maven projects for test wars already built at test1.zip and test2.zip. If you download them and extract them to `/Downloads/test1` and `/Downloads/test2`, you can see that `/Downloads/test1/target/test1.war` contains a `META-INF/cool.txt` while `/Downloads/test2/target/test2.war` does not contain that file. From CLI deploy `test1.war` first:

```
[standalone@localhost:9999 /] deploy ~/Downloads/test1/target/test1.war
'test1.war' deployed successfully.
```

And you should now see the output from the war `TrackerService` list the deployments:

```
15:35:03,712 INFO  [org.jboss.as.server.deployment] (MSC service thread 1-2) Starting deployment
of "test1.war"
15:35:03,988 INFO  [org.jboss.web] (MSC service thread 1-1) registering web context: /test1
15:35:03,996 INFO  [org.jboss.as.server.controller] (pool-2-thread-9) Deployed "test1.war"
15:35:13,056 INFO  [stdout] (Thread-9) Current deployments deployed while war tracking active:
15:35:13,056 INFO  [stdout] (Thread-9) [test1.war]
15:35:13,057 INFO  [stdout] (Thread-9) Cool: 1
```

So our `test1.war` got picked up as a 'cool' deployment. Now if we deploy `test2.war`

```
[standalone@localhost:9999 /] deploy ~/sourcecontrol/temp/archetype-test/test2/target/test2.war
'test2.war' deployed successfully.
```

You will see that deployment get picked up as well but since there is no `META-INF/cool.txt` it is not marked as a 'cool' deployment:

```
15:37:05,634 INFO  [org.jboss.as.server.deployment] (MSC service thread 1-4) Starting deployment
of "test2.war"
15:37:05,699 INFO  [org.jboss.web] (MSC service thread 1-1) registering web context: /test2
15:37:05,982 INFO  [org.jboss.as.server.controller] (pool-2-thread-15) Deployed "test2.war"
15:37:13,075 INFO  [stdout] (Thread-9) Current deployments deployed while war tracking active:
15:37:13,075 INFO  [stdout] (Thread-9) [test1.war, test2.war]
15:37:13,076 INFO  [stdout] (Thread-9) Cool: 1
```

An undeploy

```
[standalone@localhost:9999 /] undeploy test1.war
Successfully undeployed test1.war.
```

is also reflected in the `TrackerService` output:

```
15:38:47,901 INFO  [org.jboss.as.server.controller] (pool-2-thread-21) Undeployed "test1.war"
15:38:47,934 INFO  [org.jboss.as.server.deployment] (MSC service thread 1-3) Stopped deployment
test1.war in 40ms
15:38:53,091 INFO  [stdout] (Thread-9) Current deployments deployed while war tracking active:
15:38:53,092 INFO  [stdout] (Thread-9) [test2.war]
15:38:53,092 INFO  [stdout] (Thread-9) Cool: 0
```

Finally, we registered a write attribute handler for the `tick` property of the `type` so we can change the frequency

```
[standalone@localhost:9999 /] /subsystem=tracker/type=war:write-attribute(name=tick,value=1000)
{"outcome" => "success"}
```

You should now see the output from the `TrackerService` happen every second

```
15:39:43,100 INFO  [stdout] (Thread-9) Current deployments deployed while war tracking active:
15:39:43,100 INFO  [stdout] (Thread-9) [test2.war]
15:39:43,101 INFO  [stdout] (Thread-9) Cool: 0
15:39:44,101 INFO  [stdout] (Thread-9) Current deployments deployed while war tracking active:
15:39:44,102 INFO  [stdout] (Thread-9) [test2.war]
15:39:44,105 INFO  [stdout] (Thread-9) Cool: 0
15:39:45,106 INFO  [stdout] (Thread-9) Current deployments deployed while war tracking active:
15:39:45,106 INFO  [stdout] (Thread-9) [test2.war]
```

If you open `$WFLY/standalone/configuration/standalone.xml` you can see that our subsystem entry reflects the current state of the subsystem:

```
<subsystem xmlns="urn:com.acme.corp.tracker:1.0">
            <deployment-types>
                <deployment-type suffix="war" tick="1000"/>
            </deployment-types>
        </subsystem>
```

11.3.7 Expressions

Expressions are mechanism that enables you to support variables in your attributes, for instance when you want the value of attribute to be resolved using system / environment properties.

An example expression is

```
${jboss.bind.address.management:127.0.0.1}
```

which means that the value should be taken from a system property named `jboss.bind.address.management` and if it is not defined use `127.0.0.1`.

What expression types are supported

- System properties, which are resolved using `java.lang.System.getProperty(String key)`
- Environment properties, which are resolved using `java.lang.System.getEnv(String name)`.
- Security vault expressions, resolved against the security vault configured for the server or Host Controller that needs to resolve the expression.

In all cases, the syntax for the expression is

```
${expression_to_resolve}
```

For an expression meant to be resolved against environment properties, the `expression_to_resolve` must be prefixed with `env.`. The portion after `env.` will be the name passed to `java.lang.System.getEnv(String name)`.

Security vault expressions do not support default values (i.e. the `127.0.0.1` in the `jboss.bind.address.management:127.0.0.1` example above.)

How to support expressions in subsystems

The easiest way is by using AttributeDefinition, which provides support for expressions just by using it correctly.

When we create an AttributeDefinition all we need to do is mark that is allows expressions. Here is an example how to define an attribute that allows expressions to be used.

```
SimpleAttributeDefinition MY_ATTRIBUTE =
            new SimpleAttributeDefinitionBuilder("my-attribute", ModelType.INT, true)
                    .setAllowExpression(true)
                    .setFlags(AttributeAccess.Flag.RESTART_ALL_SERVICES)
                    .setDefaultValue(new ModelNode(1))
                    .build();
```

Then later when you are parsing the xml configuration you should use the MY_ATTRIBUTE attribute definition to set the value to the management operation ModelNode you are creating.

```
....
      String attr = reader.getAttributeLocalName(i);
      String value = reader.getAttributeValue(i);
      if (attr.equals("my-attribute")) {
          MY_ATTRIBUTE.parseAndSetParameter(value, operation, reader);
      } else if (attr.equals("suffix")) {
.....
```

Note that this just helps you to properly set the value to the model node you are working on, so no need to additionally set anything to the model for this attribute. Method parseAndSetParameter parses the value that was read from xml for possible expressions in it and if it finds any it creates special model node that defines that node is of type `ModelType.EXPRESSION`.

Later in your operation handlers where you implement populateModel and have to store the value from the operation to the configuration model you also use this MY_ATTRIBUTE attribute definition.

```
@Override
 protected void populateModel(ModelNode operation, ModelNode model) throws
OperationFailedException {
        MY_ATTRIBUTE.validateAndSet(operation,model);
 }
```

This will make sure that the attribute that is stored from the operation to the model is valid and nothing is lost. It also checks the value stored in the operation `ModelNode`, and if it isn't already `ModelType.EXPRESSION`, it checks if the value is a string that contains the expression syntax. If so, the value stored in the model will be of type `ModelType.EXPRESSION`. Doing this ensures that expressions are properly handled when they appear in operations that weren't created by the subsystem parser, but are instead passed in from CLI or admin console users.

As last step we need to use the value of the attribute. This is usually needed inside of the `performRuntime` method

```
protected void performRuntime(OperationContext context, ModelNode operation, ModelNode model,
ServiceVerificationHandler verificationHandler, List<ServiceController<?>> newControllers)
throws OperationFailedException {
       ....
        final int attributeValue = MY_ATTRIBUTE.resolveModelAttribute(context,
model).asInt();
        ...

    }
```

As you can see resolving of attribute's value is not done until it is needed for use in the subsystem's runtime services. The resolved value is not stored in the configuration model, the unresolved expression is. That way we do not lose any information in the model and can assure that also marshalling is done properly, where we must marshall back the unresolved value.

Attribute definitinon also helps you with that:

```
public void writeContent(XMLExtendedStreamWriter writer, SubsystemMarshallingContext context)
throws XMLStreamException {
    ....
      MY_ATTRIBUTE.marshallAsAttribute(sessionData, writer);
      MY_OTHER_ATTRIBUTE.marshallAsElement(sessionData, false, writer);
    ...
}
```

11.4 Working with WildFly Capabilities

An extension to WildFly will likely want to make use of services provided by the WildFly kernel, may want to make use of services provided by other subsystems, and may wish to make functionality available to other extensions. Each of these cases involves integration between different parts of the system. In releases prior to WildFly 10, this kind of integration was done on an ad-hoc basis, resulting in overly tight coupling between different parts of the system and overly weak integration contracts. For example, a service installed by subsystem A might depend on a service installed by subsystem B, and to record that dependency A's authors copy a ServiceName from B's code, or even refer to a constant or static method from B's code. The result is B's code cannot evolve without risking breaking A. And the authors of B may not even intend for other subsystems to use its services. There is no proper integration contract between the two subsystems.

Beginning with WildFly Core 2 and WildFly 10 the WildFly kernel's management layer provides a mechanism for allowing different parts of the system to integrate with each other in a loosely coupled manner. This is done via WildFly Capabilities. Use of capabilities provides the following benefits:

1. A standard way for system components to define integration contracts for their use by other system components.
2. A standard way for system components to access integration contracts provided by other system components.
3. A mechanism for configuration model referential integrity checking, such that if one component's configuration has an attribute that refers to an other component (e.g. a `socket-binding` attribute in a subsystem that opens a socket referring to that socket's configuration), the validity of that reference can be checked when validating the configuration model.

11.4.1 Capabilities

A capability is a piece of functionality used in a WildFly Core based process that is exposed via the WildFly Core management layer. Capabilities may depend on other capabilities, and this interaction between capabilities is mediated by the WildFly Core management layer.

Some capabilities are automatically part of a WildFly Core based process, but in most cases the configuration provided by the end user (i.e. in standalone.xml, domain.xml and host.xml) determines what capabilities are present at runtime. It is the responsibility of the handlers for management operations to register capabilities and to register any requirements those capabilities may have for the presence of other capabilities. This registration is done during the MODEL stage of operation execution

A capability has the following basic characteristics:

1. It has a name.
2. It may install an MSC service that can be depended upon by services installed by other capabilities. If it does, it provides a mechanism for discovering the name of that service.
3. It may expose some other API not based on service dependencies allowing other capabilities to integrate with it at runtime.
4. It may depend on, or **require** other capabilities.

During boot of the process, and thereafter whenever a management operation makes a change to the process' configuration, at the end of the MODEL stage of operation execution the kernel management layer will validate that all capabilities required by other capabilities are present, and will fail any management operation step that introduced an unresolvable requirement. This will be done before execution of the management operation proceeds to the RUNTIME stage, where interaction with the process' MSC Service Container is done. As a result, in the RUNTIME stage the handler for an operation can safely assume that the runtime services provided by a capability for which it has registered a requirement are available.

Comparison to other concepts

Capabilities vs modules

A JBoss Modules module is the means of making resources available to the classloading system of a WildFly Core based process. To make a capability available, you must package its resources in one or more modules and make them available to the classloading system. But a module is not a capability in and of itself, and simply copying a module to a WildFly installation does not mean a capability is available. Modules can include resources completely unrelated to management capabilities.

Capabilities vs Extensions

An extension is the means by which the WildFly Core management layer is made aware of manageable functionality that is not part of the WildFly Core kernel. The extension registers with the kernel new management resource types and handlers for operations on those resources. One of the things a handler can do is register or unregister a capability and its requirements. An extension may register a single capability, multiple capabilities, or possibly none at all. Further, not all capabilities are registered by extensions; the WildFly Core kernel itself may register a number of different capabilities.

Capability Names

Capability names are simple strings, with the dot character serving as a separator to allow namespacing.

The 'org.wildfly' namespace is reserved for projects associated with the WildFly organization on github (https://github.com/wildfly).

Statically vs Dynamically Named Capabilities

The full name of a capability is either statically known, or it may include a statically known base element and then a dynamic element. The dynamic part of the name is determined at runtime based on the address of the management resource that registers the capability. For example, the management resource at the address '/socket-binding-group=standard-sockets/socket-binding=web' will register a dynamically named capability named 'org.wildlfy.network.socket-binding.web'. The 'org.wildlfy.network.socket-binding' portion is the static part of the name.

All dynamically named capabilities that have the same static portion of their name should provide a consistent feature set and set of requirements.

Service provided by a capability

Typically a capability functions by registering a service with the WildFly process' MSC ServiceContainer, and then dependent capabilities depend on that service. The WildFly Core management layer orchestrates registration of those services and service dependencies by providing a means to discover service names.

Custom integration APIs provided by a capability

Instead of or in addition to providing MSC services, a capability may expose some other API to dependent capabilities. This API must be encapsulated in a single class (although that class can use other non-JRE classes as method parameters or return types).

Capability Requirements

A capability may rely on other capabilities in order to provide its functionality at runtime. The management operation handlers that register capabilities are also required to register their requirements.

There are three basic types of requirements a capability may have:

- Hard requirements. The required capability must always be present for the dependent capability to function.
- Optional requirements. Some aspect of the configuration of the dependent capability controls whether the depended on capability is actually necessary. So the requirement cannot be known until the running configuration is analyzed.
- Runtime-only requirements. The dependent capability will check for the presence of the depended upon capability at runtime, and if present it will utilize it, but if it is not present it will function properly without the capability. There is nothing in the dependent capability's configuration that controls whether the depended on capability must be present. Only capabilities that declare themselves as being suitable for use as a runtime-only requirement should be depended upon in this manner.

Hard and optional requirements may be for either statically named or dynamically named capabilities. Runtime-only requirements can only be for statically named capabilities, as such a requirement cannot be specified via configuration, and without configuration the dynamic part of the required capability name is unknown.

Supporting runtime-only requirements

Not all capabilities are usable as a runtime-only requirement.

Any dynamically named capability is not usable as a runtime-only requirement.

For a capability to support use as a runtime-only requirement, it must guarantee that a configuration change to a running process that removes the capability will not impact currently running capabilities that have a runtime-only requirement for it. This means:

- A capability that supports runtime-only usage must ensure that it never removes its runtime service except via a full process reload.
- A capability that exposes a custom integration API generally is not usable as a runtime-only requirement. If such a capability does support use as a runtime-only requirement, it must ensure that any functionality provided via its integration API remains available as long as a full process reload has not occurred.

11.4.2 Capability Contract

A capability provides a stable contract to users of the capability. The contract includes the following:

- The name of the capability (including whether it is dynamically named).
- Whether it installs an MSC Service, and if it does, the value type of the service. That value type then becomes a stable API users of the capability can rely upon.
- Whether it provides a custom integration API, and if it does, the type that represents that API. That type then becomes a stable API users of the capability can rely upon.
- Whether the capability supports use as a runtime-only requirement.

Developers can learn about available capabilities and the contracts they provide by reading the WildFly *capabilty registry*.

11.4.3 Capability Registry

The WildFly organization on github maintains a git repo where information about available capabilities is published.

https://github.com/wildfly/wildfly-capabilities

Developers can learn about available capabilities and the contracts they provide by reading the WildFly capabilty registry.

The README.md file at the root of that repo explains the how to find out information about the registry.

Developers of new capabilities are **strongly encouraged** to document and register their capability by submitting a pull request to the wildfly-capabilities github repo. This both allows others to learn about your capability and helps prevent capability name collisions. Capabilities that are used in the WildFly or WildFly Core code base itself **must** have a registry entry before the code referencing them will be merged.

External organizations that create capabilities should include an organization-specific namespace as part their capability names to avoid name collisions.

11.4.4 Using Capabilities

Now that all the background information is presented, here are some specifics about how to use WildFly capabilities in your code.

Basics of Using Your Own Capability

Creating your capability

A capability is an instance of the immutable `org.jboss.as.controller.capability.RuntimeCapability` class. A capability is usually registered by a resource, so the usual way to use one is to store it in constant in the resource's `ResourceDefinition`. Use a `RuntimeCapability.Builder` to create one.

```
class MyResourceDefinition extends SimpleResourceDefinition {

    static final RuntimeCapability<Void> FOO_CAPABILITY =
RuntimeCapability.Builder.of("com.example.foo").build();

    . . .
}
```

That creates a statically named capability named `com.example.foo`.

If the capability is dynamically named, add the `dynamic` parameter to state this:

```
static final RuntimeCapability<Void> FOO_CAPABILITY =
            RuntimeCapability.Builder.of("com.example.foo", true).build();
```

Most capabilities install a service that requiring capabilities can depend on. If your capability does this, you need to declare the service's *value type* (the type of the object returned by `org.jboss.msc.Service.getValue()`). For example, if FOO_CAPABILITY provides a `Service<javax.sql.DataSource>`:

```
static final RuntimeCapability<Void> FOO_CAPABILITY =
            RuntimeCapability.Builder.of("com.example.foo", DataSource.class).build();
```

For a dynamic capability:

```
static final RuntimeCapability<Void> FOO_CAPABILITY =
           RuntimeCapability.Builder.of("com.example.foo", true, DataSource.class).build();
```

If the capability provides a custom integration API, you need to instantiate an instance of that API:

```
public class JTSCapability {

    static final JTSCapability INSTANCE = new JTSCapability();

    private JTSCapability() {}

    /**
     * Gets the names of the {@link org.omg.PortableInterceptor.ORBInitializer} implementations
that should be included
     * as part of the {@link org.omg.CORBA.ORB#init(String[], java.util.Properties)
initialization of an ORB}.
     *
     * @return the names of the classes implementing {@code ORBInitializer}. Will not be {@code
null}.
     */
    public List<String> getORBInitializerClasses() {
        return Collections.unmodifiableList(Arrays.asList(

"com.arjuna.ats.jts.orbspecific.jacorb.interceptors.interposition.InterpositionORBInitializerImpl"
"com.arjuna.ats.jbossatx.jts.InboundTransactionCurrentInitializer"));
    }
}
```

and provide it to the builder:

```
static final RuntimeCapability<JTSCapability> FOO_CAPABILITY =
            RuntimeCapability.Builder.of("com.example.foo", JTSCapability.INSTANCE).build();
```

For a dynamic capability:

```
static final RuntimeCapability<JTSCapability> FOO_CAPABILITY =
RuntimeCapability.Builder.of("com.example.foo", true, JTSCapability.INSTANCE).build();
```

A capability can provide both a custom integration API and install a service:

```
static final RuntimeCapability<JTSCapability> FOO_CAPABILITY =
            RuntimeCapability.Builder.of("com.example.foo", JTSCapability.INSTANCE)
                .setServiceType(DataSource.class)
                .build();
```

Registering and unregistering your capability

Once you have your capability, you need to ensure it gets registered with the WildFly Core kernel when your resource is added. This is easily done simply by providing a reference to the capability to the resource's `ResourceDefinition`. This assumes your resource definition is a subclass of the standard `org.jboss.as.controller.SimpleResourceDefinition`. `SimpleResourceDefinition` provides a `Parameters` class that provides a builder-style API for setting up all the data needed by your definition. This includes a `setCapabilities` method that can be used to declare the capabilities provided by resources of this type.

```
class MyResourceDefinition extends SimpleResourceDefinition {

    . . .

    MyResourceDefinition() {
        super(new SimpleResourceDefinition.Parameters(PATH, RESOLVER)
            .setAddHandler(MyAddHandler.INSTANCE)
            .setRemoveHandler(MyRemoveHandler.INSTANCE)
            .setCapabilities(FOO_CAPABILITY)
            );
    }
}
```

Your add handler needs to extend the standard `org.jboss.as.controller.AbstractAddStepHandler` class or one of its subclasses:

```
class MyAddHandler extends AbstractAddStepHandler() {
```

`AbstractAddStepHandler`'s logic will register the capability when it executes.

Your remove handler must also extend of the standard `org.jboss.as.controller.AbstractRemoveStepHandler` or one of its subclasses.

```
class MyRemoveHandler extends AbstractRemoveStepHandler() {
```

`AbstractRemoveStepHandler`'s logic will deregister the capability when it executes.

If for some reason you cannot base your `ResourceDefinition` on `SimpleResourceDefinition` or your handlers on `AbstractAddStepHandler` and `AbstractRemoveStepHandler` then you will need to take responsibility for registering the capability yourself. This is not expected to be a common situation. See the implementation of those classes to see how to do it.

Installing, accessing and removing the service provided by your capability

If your capability installs a service, you should use the `RuntimeCapability` when you need to determine the service's name. For example in the `Stage.RUNTIME` handling of your "add" step handler. Here's an example for a statically named capability:

```
class MyAddHandler extends AbstractAddStepHandler() {

    . . .

    @Override
    protected void performRuntime(final OperationContext context, final ModelNode operation,
                                  final Resource resource) throws OperationFailedException {

        ServiceName serviceName = FOO_CAPABILITY.getCapabilityServiceName();
        Service<DataSource> service = createDataSourceService(context, resource);
        context.getServiceTarget().addService(serviceName, service).install();

    }
```

If the capability is dynamically named, get the dynamic part of the name from the `OperationContext` and use that when getting the service name:

```
class MyAddHandler extends AbstractAddStepHandler() {

    . . .

    @Override
    protected void performRuntime(final OperationContext context, final ModelNode operation,
                                  final Resource resource) throws OperationFailedException {

        String myName = context.getCurrentAddressValue();
        ServiceName serviceName = FOO_CAPABILITY.getCapabilityServiceName(myName);
        Service<DataSource> service = createDataSourceService(context, resource);
        context.getServiceTarget().addService(serviceName, service).install();

    }
```

The same patterns should be used when accessing or removing the service in handlers for `remove`, `write-attribute` and custom operations.

If you use `ServiceRemoveStepHandler` for the `remove` operation, simply provide your `RuntimeCapability` to the `ServiceRemoveStepHandler` constructor and it will automatically remove your capability's service when it executes.

Basics of Using Other Capabilities

When a capability needs another capability, it only refers to it by its string name. A capability should not reference the `RuntimeCapability` object of another capability.

Before a capability can look up the service name for a required capability's service, or access its custom integration API, it must first register a requirement for the capability. This must be done in Stage.MODEL, while service name lookups and accessing the custom integration API is done in Stage.RUNTIME.

Registering a requirement for a capability is simple.

Registering a hard requirement for a static capability

If your capability has a hard requirement for a statically named capability, simply declare that to the builder for your `RuntimeCapability`. For example, WildFly's JTS capability requires both a basic transaction support capability and IIOP capabilities:

```
static final RuntimeCapability<JTSCapability> JTS_CAPABILITY =
            RuntimeCapability.Builder.of("org.wildfly.transactions.jts", new JTSCapability())
                .addRequirements("org.wildfly.transactions", "org.wildfly.iiop.orb",
"org.wildfly.iiop.corba-naming")
                .build();
```

When your capability is registered with the system, the WildFly Core kernel will automatically register any static hard requirements declared this way.

Registering a requirement for a dynamically named capability

If the capability you require is dynamically named, usually your capability's resource will include an attribute whose value is the dynamic part of the required capability's name. You should declare this fact in the `AttributeDefinition` for the attribute using the `SimpleAttributeDefinitionBuilder.setCapabilityReference` method.

For example, the WildFly "remoting" subsystem's "org.wildfly.remoting.connector" capability has a requirement for a dynamically named socket-binding capability:

```
public class ConnectorResource extends SimpleResourceDefinition {

    . . .

    static final String SOCKET_CAPABILITY_NAME = "org.wildfly.network.socket-binding";
    static final RuntimeCapability<Void> CONNECTOR_CAPABILITY =
            RuntimeCapability.Builder.of("org.wildfly.remoting.connector", true)
                    .build();

    . . .

    static final SimpleAttributeDefinition SOCKET_BINDING =
            new SimpleAttributeDefinitionBuilder(CommonAttributes.SOCKET_BINDING,
ModelType.STRING, false)

.addAccessConstraint(SensitiveTargetAccessConstraintDefinition.SOCKET_BINDING_REF)
                .setCapabilityReference(SOCKET_CAPABILITY_NAME, CONNECTOR_CAPABILITY)
                .build();
```

If the "add" operation handler for your resource extends `AbstractAddStepHandler` and the handler for `write-attribute` extends `AbstractWriteAttributeHandler`, the declaration above is sufficient to ensure that the appropriate capability requirement will be registered when the attribute is modified.

Depending upon a service provided by another capability

Once the requirement for the capability is registered, your `OperationStepHandler` can use the `OperationContext` to discover the name of the service provided by the required capability.

For example, the "add" handler for a remoting connector uses the `OperationContext` to find the name of the needed {{SocketBinding} service:

```
final String socketName = ConnectorResource.SOCKET_BINDING.resolveModelAttribute(context,
fullModel).asString();
        final ServiceName socketBindingName =
context.getCapabilityServiceName(ConnectorResource.SOCKET_CAPABILITY_NAME, socketName,
SocketBinding.class);
```

That service name is then used to add a dependency on the `SocketBinding` service to the remoting connector service.

If the required capability isn't dynamically named, `OperationContext` exposes an overloaded `getCapabilityServiceName` variant. For example, if a capability requires a remoting Endpoint:

```
ServiceName endpointService = context.getCapabilityServiceName("org.wildfly.remoting.endpoint",
Endpoint.class);
```

Using a custom integration API provided by another capability

In your `Stage.RUNTIME` handler, use `OperationContext.getCapabilityRuntimeAPI` to get a reference to the required capability's custom integration API. Then use it as necessary.

```
List<String> orbInitializers = new ArrayList<String>();
        . . .
        JTSCapability jtsCapability =
context.getCapabilityRuntimeAPI(IIOPExtension.JTS_CAPABILITY, JTSCapability.class);
        orbInitializers.addAll(jtsCapability.getORBInitializerClasses());
```

Runtime-only requirements

If your capability has a runtime-only requirement for another capability, that means that if that capability is present in `Stage.RUNTIME` you'll use it, and if not you won't. There is nothing about the configuration of your capability that triggers the need for the other capability; you'll just use it if it's there.

In this case, use `OperationContext.hasOptionalCapability` in your `Stage.RUNTIME` handler to check if the capability is present:

```
protected void performRuntime(final OperationContext context, final ModelNode operation, final
ModelNode model) throws OperationFailedException {

        ServiceName myServiceName = MyResource.FOO_CAPABILITY.getCapabilityServiceName();
        Service<DataSource> myService = createService(context, model);
        ServiceBuilder<DataSource> builder = context.getTarget().addService(myServiceName,
myService);

        // Inject a "Bar" into our "Foo" if bar capability is present
        if (context.hasOptionalCapability("com.example.bar",
MyResource.FOO_CAPABILITY.getName(), null) {
            ServiceName barServiceName = context.getCapabilityServiceName("com.example.bar",
Bar.class);
            builder.addDependency(barServiceName, Bar.class, myService.getBarInjector());
        }

        builder.install();
    }
```

The WildFly Core kernel will not register a requirement for the "com.example.bar" capability, so if a configuration change occurs that means that capability will no longer be present, that change will not be rolled back. Because of this, runtime-only requirements can only be used with capabilities that declare in their contract that they support such use.

Using a capability in a DeploymentUnitProcessor

A `DeploymentUnitProcessor` is likely to have a need to interact with capabilities, in order to create service dependencies from a deployment service to a capability provided service or to access some aspect of a capability's custom integration API that relates to deployments.

If a `DeploymentUnitProcessor` associated with a capability implementation needs to utilize its own capability object, the `DeploymentUnitProcessor` authors should simply provide it with a reference to the `RuntimeCapability` instance. Service name lookups or access to the capabilities custom integration API can then be performed by invoking the methods on the `RuntimeCapability`.

If you need to access service names or a custom integration API associated with a different capability, you will need to use the `org.jboss.as.controller.capability.CapabilityServiceSupport` object associated with the deployment unit. This can be found as an attachment to the `DeploymentPhaseContext`:

```
class MyDUP implements DeploymentUntiProcessor {

    public void deploy(DeploymentPhaseContext phaseContext) throws
DeploymentUnitProcessingException {

        AttachmentKey<CapabilityServiceSupport> key =
org.jboss.as.server.deployment.Attachments.DEPLOYMENT_COMPLETE_SERVICES;
        CapabilityServiceSupport capSvcSupport = phaseContext.getAttachment(key);
```

Once you have the `CapabilityServiceSupport` you can use it to look up service names:

```
ServiceName barSvcName = capSvcSupport.getCapabilityServiceName("com.example.bar");
        // Determine what 'baz' the user specified in the deployment descriptor
        String bazDynamicName = getSelectedBaz(phaseContext);
        ServiceName bazSvcName = capSvcSupport.getCapabilityServiceName("com.example.baz",
bazDynamicName);
```

It's important to note that when you request a service name associated with a capability, the `CapabilityServiceSupport` will give you one regardless of whether the capability is actually registered with the kernel. If the capability isn't present, any service dependency your DUP creates using that service name will eventually result in a service start failure, due to the missing dependency. This behavior of not failing immediately when the capability service name is requested is deliberate. It allows deployment operations that use the `rollback-on-runtime-failure=false` header to successfully install (but not start) all of the services related to a deployment. If a subsequent operation adds the missing capability, the missing service dependency problem will then be resolved and the MSC service container will automatically start the deployment services.

You can also use the `CapabilityServiceSupport` to obtain a reference to the capability's custom integration API:

```
// We need custom integration with the baz capability beyond service injection
        BazIntegrator bazIntegrator;
        try {
            bazIntegrator = capSvcSupport.getCapabilityRuntimeAPI("com.example.baz",
bazDynamicName, BazIntegrator.class);
        } catch (NoSuchCapabilityException e) {
            //
            String msg = String.format("Deployment %s requires use of the 'bar' capability but
it is not currently registered",
                                       phaseContext.getDeploymentUnit().getName());
            throw new DeploymentUnitProcessingException(msg);
        }
```

Note that here, unlike the case with service name lookups, the `CapabilityServiceSupport` will throw a checked exception if the desired capability is not installed. This is because the kernel has no way to satisfy the request for a custom integration API if the capability is not installed. The `DeploymentUnitProcessor` will need to catch and handle the exception.

Detailed API

The WildFly Core kernel's API for using capabilities is covered in detail in the javadoc for the RuntimeCapability and RuntimeCapability.Builder classes and the OperationContext and CapabilityServiceSupport interfaces.

Many of the methods in `OperationContext` related to capabilities have to do with registering capabilities or registering requirements for capabilities. Typically non-kernel developers won't need to worry about these, as the abstract `OperationStepHandler` implementations provided by the kernel take care of this for you, as described in the preceding sections. If you do find yourself in a situation where you need to use these in an extension, please read the javadoc thoroughly.

11.5 Domain mode subsystem transformers

11.5.1 Abstract

A WildFly domain may consist of a new Domain Controller (DC) controlling slave Host Controllers (HC) running older versions. Each slave HC maintains a copy of the centralized domain configuration, which they use for controlling their own servers. In order for the slave HCs to understand the configuration from the DC, transformation is needed, whereby the DC translates the configuration and operations into something the slave HCs can understand.

11.5.2 Background

WildFly comes with a domain mode which allows you to have one Host Controller acting as the Domain Controller. The Domain Controller's job is to maintain the centralized domain configuration. Another term for the DC is 'Master Host Controller'. Before explaining why transformers are important and when they should be used, we will revisit how the domain configuration is used in domain mode.

The centralized domain configuration is stored in `domain.xml`. This is only ever parsed on the DC, and it has the following structure:

- `extensions` - contains:
  - `extension` - a references to a module that bootstraps the `org.jboss.as.controller.Extension` implementation used to bootstrap your subsystem parsers and initialize the resource definitions for your subsystems.
- `profiles` - contains:
  - `profile` - a named set of:
    - `subsystem` - contains the configuration for a subsystem, using the parser initialized by the subsystem's extension.
- `socket-binding-groups` - contains:
  - `socket-binding-group` - a named set of:
    - `socket-binding` - A named port on an interface which can be referenced from the `subsystem` configurations for subsystems opening sockets.
- `server-groups` - contains
  - `server-group` - this has a name and references a `profile` and a `socket-binding-group`. The HCs then reference the `server-group` name from their `<servers>` section in `host.xml`.

When the DC parses `domain.xml`, it is transformed into `add` (and in some cases `write-attribute`) operations just as explained in Parsing and marshalling of the subsystem xml. These operations build up the model on the DC.

A HC wishing to join the domain and use the DC's centralized configuration is known as a 'slave HC'. A slave HC maintains a copy of the DC's centralized domain configuration. This copy of the domain configuration is used to start its servers. This is done by asking the domain model to `describe` itself, which in turn asks the subsystems to `describe` themselves. The `describe` operation for a subsystem looks at the state of the subsystem model and produces the `add` operations necessary to create the subsystem on the server. The same mechanism also takes place on the DC (bear in mind that the DC is also a HC, which can have its own servers), although of course its copy of the domain configuration is the centralized one.

There are two steps involved in keeping the keeping the slave HC's domain configuration in sync with the centralized domain configuration.

- getting the initial domain model
- an operation changes something in the domain configuration

Let's look a bit closer at what happens in each of these steps.

Getting the initial domain model

When a slave HC connects to the DC it obtains a copy of the domain model from the DC. This is done in a simpler serialized format, different from the operations that built up the model on the DC, or the operations resulting from the `describe` step used to bootstrap the servers. They describe each address that exists in the DC's model, and contain the attributes set for the resource at that address. This serialized form looks like this:

```
[{
    "domain-resource-address" => [],
    "domain-resource-model" => {
        "management-major-version" => 2,
        "management-minor-version" => 0,
        "management-micro-version" => 0,
        "release-version" => "8.0.0.Beta1-SNAPSHOT",
        "release-codename" => "WildFly"
    }
},
{
    "domain-resource-address" => [("extension" => "org.jboss.as.clustering.infinispan")],
    "domain-resource-model" => {"module" => "org.jboss.as.clustering.infinispan"}
},
--SNIP - the rest of the extensions --
{
    "domain-resource-address" => [("extension" => "org.jboss.as.weld")],
    "domain-resource-model" => {"module" => "org.jboss.as.weld"}
},
{
    "domain-resource-address" => [("system-property" => "java.net.preferIPv4Stack")],
    "domain-resource-model" => {
        "value" => "true",
        "boot-time" => undefined
    }
},
{
    "domain-resource-address" => [("profile" => "full-ha")],
    "domain-resource-model" => undefined
},
{
    "domain-resource-address" => [
        ("profile" => "full-ha"),
        ("subsystem" => "logging")
    ],
    "domain-resource-model" => {}
},
{
    "domain-resource-address" => [sss|WFLY8:Example subsystem],
    "domain-resource-model" => {
        "level" => "INFO",
        "enabled" => undefined,
        "encoding" => undefined,
        "formatter" => "%d{HH:mm:ss,SSS} %-5p [%c] (%t) %s%E%n",
        "filter-spec" => undefined,
        "autoflush" => undefined,
        "target" => undefined,
        "named-formatter" => undefined
    }
},
--SNIP---
```

The slave HC then applies these one at a time and builds up the initial domain model. It needs to do this before it can start any of its servers.

An operation changes something in the domain configuration

Once a domain is up and running we can still change things in the domain configuration. These changes must happen when connected to the DC, and are then propagated to the slave HCs, which then in turn propagate the changes to any servers running in a server group affected by the changes made. In this example:

```
[disconnected /] connect
[domain@localhost:9990 /]
/profile=full/subsystem=datasources/data-source=ExampleDS:write-attribute(name=enabled,value=false
"outcome" => "success",
    "result" => undefined,
    "server-groups" => {"main-server-group" => {"host" => {
        "slave" => {"server-one" => {"response" => {
            "outcome" => "success",
            "result" => undefined,
            "response-headers" => {
                "operation-requires-restart" => true,
                "process-state" => "restart-required"
            }
        }}},
        "master" => {
            "server-one" => {"response" => {
                "outcome" => "success",
                "response-headers" => {
                    "operation-requires-restart" => true,
                    "process-state" => "restart-required"
                }
            }},
            "server-two" => {"response" => {
                "outcome" => "success",
                "response-headers" => {
                    "operation-requires-restart" => true,
                    "process-state" => "restart-required"
                }
            }}
        }
    }}}
}
```

the DC propagates the changes to itself `host=master`, which in turn propagates it to its two servers belonging to `main-server-group` which uses the `full` profile. More interestingly, it also propagates it to `host=slave` which updates its local copy of the domain model, and then propagates the change to its `server-one` which belongs to `main-server-group` which uses the `full` profile.

11.5.3 Versions and backward compatibility

A HC and its servers will always be the same version of WildFly (they use the same module path and jars). However, the DC and the slave HCs do not necessarily need to be the same version. One of the points in the original specification for WildFly is that

Important

A Domain Controller should be able to manage slave Host Controllers older than itself.

This means that for example a WildFly 10.1 DC should be able to work with slave HCs running WildFly 10. The opposite is not true, the DC must be the same or the newest version in the domain.

Versioning of subsystems

To help with being able to know what is compatible we have versions within the subsystems, this is stored in the subsystem's extension. When registering the subsystem you will typically see something like:

```
public class SomeExtension implements Extension {

    private static final String SUBSYSTEM_NAME = "my-subsystem"'

    private static final int MANAGEMENT_API_MAJOR_VERSION = 2;
    private static final int MANAGEMENT_API_MINOR_VERSION = 0;
    private static final int MANAGEMENT_API_MICRO_VERSION = 0;

    /**
     * {@inheritDoc}
     * @see
org.jboss.as.controller.Extension#initialize(org.jboss.as.controller.ExtensionContext)
     */
    @Override
    public void initialize(ExtensionContext context) {

        // IMPORTANT: Management API version != xsd version! Not all Management API changes
result in XSD changes
        SubsystemRegistration registration = context.registerSubsystem(SUBSYSTEM_NAME,
MANAGEMENT_API_MAJOR_VERSION,
                MANAGEMENT_API_MINOR_VERSION, MANAGEMENT_API_MICRO_VERSION);

        //Register the resource definitions
        ....
    }
    ....
}
```

Which sets the `ModelVersion` of the subsystem.

Important

Whenever something changes in the subsystem, such as:

- an attribute is added or removed from a resource
- a attribute is renamed in a resource
- an attribute has its type changed
- an attribute or operation parameter's nillable or allows expressions is changed
- an attribute or operation parameter's default value changes
- a child resource type is added or removed
- an operation is added or removed
- an operation has its parameters changed

and the current version of the subsystem has been part of a Final release of WildFly, we **must** bump the version of the subsystem.

Once it has been increased you can of course make more changes until the next Final release without more version bumps. It is also worth noting that a new WildFly release does not automatically mean a new version for the subsystem, the new version is only needed if something was changed. For example the `jaxrs` subsystem has remained on 1.0.0 for all versions of WildFly and JBoss AS 7.

You can find the `ModelVersion` of a subsystem by querying its extension:

```
domain@localhost:9990 /]
/extension=org.jboss.as.clustering.infinispan:read-resource(recursive=true)
{
    "outcome" => "success",
    "result" => {
        "module" => "org.jboss.as.clustering.infinispan",
        "subsystem" => {"infinispan" => {
            "management-major-version" => 2,
            "management-micro-version" => 0,
            "management-minor-version" => 0,
            "xml-namespaces" => [jboss:domain:infinispan:1.0",
                "urn:jboss:domain:infinispan:1.1",
                "urn:jboss:domain:infinispan:1.2",
                "urn:jboss:domain:infinispan:1.3",
                "urn:jboss:domain:infinispan:1.4",
                "urn:jboss:domain:infinispan:2.0"]
        }}
    }
}
```

11.5.4 The role of transformers

Now that we have mentioned the slave HCs registration process with the DC, and know about ModelVersions, it is time to mention that when registering with the DC, the slave HC will send across a list of all its subsystem ModelVersions. The DC maintains this information in a registry for each slave HC, so that it knows which transformers (if any) to invoke for a legacy slave. We will see how to write and register transformers later on in How do I write a transformer. Slave HCs from version 7.2.0 onwards will also include a list of resources that they ignore (see Ignoring resources on legacy hosts), and the DC will maintain this information in its registry. The DC will not send across any resources that it knows a slave ignores during the initial domain model transfer. When forwarding operations onto the slave HCs, the DC will skip forwarding those to slave HCs ignoring those resources.

There are two kinds of transformers:

- resource transformers
- operation transformers

The main function of transformers is to transform a subsystem to something that the legacy slave HC can understand, or to aggressively reject things that the legacy slave HC will not understand. Rejection, in this context, essentially means, that the resource or operation cannot safely be transformed to something valid on the slave, so the transformation fails. We will see later how to reject attributes in Rejecting attributes, and child resources in Reject child resource.

Both resource and operation transformers are needed, but take effect at different times. Let us use the `weld` subsystem, which is relatively simple, as an example. In JBoss AS 7.2.0 and lower it had a ModelVersion of 1.0.0, and its resource description was as follows:

```
{
                    "description" => "The configuration of the weld subsystem.",
                    "attributes" => {},
                    "operations" => {
                        "remove" => {
                            "operation-name" => "remove",
                            "description" => "Operation removing the weld subsystem.",
                            "request-properties" => {},
                            "reply-properties" => {}
                        },
                        "add" => {
                            "operation-name" => "add",
                            "description" => "Operation creating the weld subsystem.",
                            "request-properties" => {},
                            "reply-properties" => {}
                        }
                    },
                    "children" => {}
                },
```

In WildFly 8, it has a ModelVersion of 2.0.0 and has added two attributes, `require-bean-descriptor` and `non-portable` mode:

```
{
        "description" => "The configuration of the weld subsystem.",
        "attributes" => {
            "require-bean-descriptor" => {
                "type" => BOOLEAN,
                "description" => "If true then implicit bean archives without bean descriptor
file (beans.xml) are ignored by Weld",
                "expressions-allowed" => true,
                "nillable" => true,
                "default" => false,
                "access-type" => "read-write",
                "storage" => "configuration",
                "restart-required" => "no-services"
            },
            "non-portable-mode" => {
                "type" => BOOLEAN,
                "description" => "If true then the non-portable mode is enabled. The
non-portable mode is suggested by the specification to overcome problems with legacy
applications that do not use CDI SPI properly and may be rejected by more strict validation in
CDI 1.1.",
                "expressions-allowed" => true,
                "nillable" => true,
                "default" => false,
                "access-type" => "read-write",
                "storage" => "configuration",
                "restart-required" => "no-services"
            }
        },
        "operations" => {
            "remove" => {
                "operation-name" => "remove",
                "description" => "Operation removing the weld subsystem.",
                "request-properties" => {},
                "reply-properties" => {}
            },
            "add" => {
                "operation-name" => "add",
                "description" => "Operation creating the weld subsystem.",
                "request-properties" => {
                    "require-bean-descriptor" => {
                        "type" => BOOLEAN,
                        "description" => "If true then implicit bean archives without bean
descriptor file (beans.xml) are ignored by Weld",
                        "expressions-allowed" => true,
                        "required" => false,
                        "nillable" => true,
                        "default" => false
                    },
                    "non-portable-mode" => {
                        "type" => BOOLEAN,
                        "description" => "If true then the non-portable mode is enabled. The
non-portable mode is suggested by the specification to overcome problems with legacy
applications that do not use CDI SPI properly and may be rejected by more strict validation in
```

```
CDI 1.1.",
                        "expressions-allowed" => true,
                        "required" => false,
                        "nillable" => true,
                        "default" => false
                    }
                },
                "reply-properties" => {}
            }
        },
        "children" => {}
    }
```

In the rest of this section we will assume that we are running a DC running WildFly 8 so it will have ModelVersion 2.0.0 of the weld subsystem, and that we are running a slave using ModelVersion 1.0.0 of the weld subsystem.

Important

Transformation always takes place on the Domain Controller, and is done when sending across the initial domain model AND forwarding on operations to legacy slave HCs.

Resource transformers

When copying over the centralized domain configuration as mentioned in Getting the initial domain model, we need to make sure that the copy of the domain model is something that the servers running on the legacy slave HC understand. So if the centralized domain configuration had any of the two new attributes set, we would need to reject the transformation in the transformers. One reason for this is to keep things consistent, it doesn't look good if you connect to the slave HC and find attributes and/or child resources when doing `:read-resource` which are not there when you do `:read-resource-description`. Also, to make life easier for subsystem writers, most instances of the `describe` operation use a standard implementation which would include these attributes when creating the `add` operation for the server, which could cause problems there.

Another, more concrete example from the logging subsystem is that it allows a '`%K{...}`' in the pattern formatter which makes the formatter use color:

```
<pattern-formatter pattern="%K{level}%d{HH:mm:ss,SSS} %-5p [%c] (%t) %s%E%n"/>
```

This '`%K{...}`' however was introduced in JBoss AS < 7.1.3 (ModelVersion 1.2.0), so if that makes it across to a slave HC running an older version, the servers **will** fail to start up. So the logging extension registers transformers to strip out the '`%K{...}`' from the attribute value (leaving '`%-5p`

```
[%c]
```

`(%t) %s%E%n"`') so that the old slave HC's servers can understand it.

Rejection in resource transformers

Only slave HCs from JBoss AS 7.2.0 and newer inform the DC about their ignored resources (see Ignoring resources on legacy hosts). This means that if a transformer on the DC rejects transformation for a legacy slave HC, exactly what happens to the slave HC depends on the version of the slave HC. If the slave HC is:

- *older than 7.2.0* - the DC has no means of knowing if the slave HC has ignored the resource being rejected or not. So we log a warning on the DC, and send over the serialized part of that model anyway. If the slave HC has ignored the resource in question, it does not apply it. If the slave HC has not ignored the resource in question, it will apply it, but no failure will happen until it tries to start a server which references this bad configuration.
- *7.2.0 or newer* - If a resource is ignored on the slave HC, the DC knows about this, and will not attempt to transform or send the resource across to the slave HC. If the resource transformation is rejected, we know the resource was not ignored on the slave HC and so we can aggressively fail the transformation, which in turn will cause the slave HC to fail to start up.

Operation transformers

When An operation changes something in the domain configuration the operation gets sent across to the slave HCs to update their copies of the domain model. The slave HCs then forward this operation onto the affected servers. The same considerations as in Resource transformers are true, although operation transformers give you quicker 'feedback' if something is not valid. If you try to execute:

```
/profile=full/subsystem=weld:write-attribute(name=require-bean-descriptor, value=false)
```

This will fail on the legacy slave HC since its version of the subsystem does not contain any such attribute. However, it is best to aggressively reject in such cases.

Rejection in operation transformers

For transformed operations we can always know if the operation is on an ignored resource in the legacy slave HC. In 7.2.0 onwards, we know this through the DC's registry of ignored resources on the slave. In older versions of slaves, we send the operation across to the slave, which tries to invoke the operation. If the operation is against an ignored resource we inform the DC about this fact. So as part of the transformation process, if something gets rejected we can (and do!) fail the transformation aggressively. If the operation invoked on the DC results in the operation being sent across to 10 slave HCs and one of them has a legacy version which ends up rejecting the transformation, we rollback the operation across the whole domain.

Different profiles for different versions

Now for the `weld` example we have been using there is a slight twist. We have the new `require-bean-descriptor` and `non-portable-mode` attributes. These have been added in WildFly 8 which supports Java EE 7, and thus CDI 1.1. JBoss AS 7.x supports Java EE 6, and thus CDI 1.0. In CDI 1.1 the values of these attributes are tweakable, so they can be set to either `true` or `false`. The default behaviour for these in CDI 1.1, if not set, is that they are `false`. However, for CDI 1.0 these were not tweakable, and with the way the subsystem in JBoss AS 7.x worked is similar to if they are set to `true`.

The above discussion implies that to use the weld subsystem on a legacy slave HC, the `domain.xml` configuration for it must look like:

```
<subsystem xmlns="urn:jboss:domain:weld:2.0"
      require-bean-descriptor="true"
      non-portable-mode="true"/>
```

We will see the exact mechanics for how this is actually done later but in short when pushing this to a legacy slave DC we register transformers which reject the transformation if these attributes are not set to `true` since that implies some behavior not supported on the legacy slave DC. If they are `true`, all is well, and the transformers discard, or remove, these attributes since they don't exist in the legacy model. This removal is fine since they have the values which would result in the behavior assumed on the legacy slave HC.

That way the older slave HCs will work fine. However, we might also have WildFly 8 slave HCs in our domain, and they are missing out on the new features introduced by the attributes introduced in ModelVersion 2.0.0. If we do

```
<subsystem xmlns="urn:jboss:domain:weld:2.0"
      require-bean-descriptor="false"
      non-portable-mode="false"/>
```

then it will fail when doing transformation for the legacy controller. The solution is to put these in two different profiles in `domain.xml`

```
<domain>
....
  <profiles>
    <profile name="full">
      <subsystem xmlns="urn:jboss:domain:weld:2.0"
        require-bean-descriptor="false"
        non-portable-mode="false"/>
      ...
    </profile>
    <profile name="full-legacy">
      <subsystem xmlns="urn:jboss:domain:weld:2.0"
        require-bean-descriptor="true"
        non-portable-mode="true"/>
      ...
    </profile>
  </profiles>
  ...
  <server-groups>
    <server-group name="main-server-group" profile="full">
      ....
    <server-group>
    <server-group name="main-server-group-legacy" profile="full-legacy">
      ....
    <server-group>
  </server-groups>
</domain>
```

Then have the HCs using WildFly 8 make their servers reference the `main-server-group` server group, and the HCs using older versions of WildFly 8 make their servers reference the `main-server-group-legacy` server group.

Ignoring resources on legacy hosts

Booting the above configuration will still cause problems on legacy slave HCs, especially if they are JBoss AS 7.2.0 or later. The reason for this is that when they register themselves with the DC it lets the DC know which `ignored resources` they have. If the DC comes to transform something it should reject for a slave HC and it is not part of its ignored resources it will aggressively fail the transformation. Versions of JBoss AS older than 7.2.0 still have this ignored resources mechanism, but don't let the DC know about what they have ignored so the DC cannot reject aggressively - instead it will log some warnings. However, it is still good practice to ignore resources you are not interested in regardless of which legacy version the slave HC is running.

To ignore the profile we cannot understand we do the following in the legacy slave HC's `host.xml`

```
<host xmlns="urn:jboss:domain:1.3" name="slave">
...
    <domain-controller>
       <remote host="${jboss.test.host.master.address}" port="${jboss.domain.master.port:9999}"
security-realm="ManagementRealm">
            <ignored-resources type="profile">
                <instance name="full-legacy"/>
            </ignored-resources>
       </remote>
    </domain-controller>
....
</host>
```

Important

Any top-level resource type can be ignored `profile`, `extension`, `server-group` etc. Ignoring a resource instance ignores that resource, and all its children.

11.5.5 How do I know what needs to be transformed?

There is a set of related classes in the `org.wildfly.legacy.util` package to help you determine this. These now live at https://github.com/wildfly/wildfly-legacy-test/tree/master/tools/src/main/java/org/wildfly/legacy/util.
They are all runnable in your IDE, just start the WildFly or JBoss AS 7 instances as described below.

Getting data for a previous version

https://github.com/wildfly/wildfly-legacy-test/tree/master/tools/src/main/resources/legacy-models contains the output for the previous WildFly/JBoss AS 7 versions, so check if the files for the version you want to check backwards compatibility are there yet. If not, then you need to do the following to get the subsystem definitions:

1. Start the **old** version of WildFly/JBoss AS 7 using `--server-config=standalone-full-ha.xml`
2. Run `org.wildfly.legacy.util.GrabModelVersionsUtil`, which will output the subsystem versions to `target/standalone-model-versions-running.dmr`
3. Run `org.wildfly.legacy.util.DumpStandaloneResourceDefinitionUtil` which will output the full resource definition to `target/standalone-resource-definition-running.dmr`
4. Stop the running version of WildFly/JBoss AS 7

See what changed

To do this follow the following steps

1. Start the **new** version of WildFly using `--server-config=standalone-full-ha.xml`
2. Run `org.wildfly.legacy.util.CompareModelVersionsUtil` and answer the following questions"
    1. Enter Legacy AS version:
        - If it is known version in the `tools/src/test/resources/legacy-models` folder, enter the version number.
        - If it is a not known version, and you got the data yourself in the last step, enter '`running`'
    2. Enter type:
        - Answer 'S'
    3. Read from target directory or from the legacy-models directory:
        - If it is known version in the `controller/src/test/resources/legacy-models` folder, enter '`l`'.
        - If it is a not known version, and you got the data yourself in the last step, enter '`t`'
    4. Report on differences in the model when the management versions are different?:
        - Answer 'y'

Here is some example output, as a subsystem developer you can ignore everything down to `======` `Comparing subsystem models ======`:

```
Enter legacy AS version: 7.2.0.Final
Using target model: 7.2.0.Final
Enter type [S](standalone)/H(host)/D(domain)/F(domain + host):S
Read from target directory or from the legacy-models directory - t/[l]:
Report on differences in the model when the management versions are different? y/[n]: y
Reporting on differences in the model when the management versions are different
Loading legacy model versions for 7.2.0.Final....
Loaded legacy model versions
Loading model versions for currently running server...
Oct 01, 2013 6:26:03 PM org.xnio.Xnio <clinit>
INFO: XNIO version 3.1.0.CR7
Oct 01, 2013 6:26:03 PM org.xnio.nio.NioXnio <clinit>
INFO: XNIO NIO Implementation Version 3.1.0.CR7
Oct 01, 2013 6:26:03 PM org.jboss.remoting3.EndpointImpl <clinit>
INFO: JBoss Remoting version 4.0.0.Beta1
Loaded current model versions
Loading legacy resource descriptions for 7.2.0.Final....
Loaded legacy resource descriptions
Loading resource descriptions for currently running STANDALONE...
Loaded current resource descriptions
Starting comparison of the current....

====== Comparing core models ======
-- SNIP --

====== Comparing subsystem models ======
-- SNIP --
====== Resource root address: ["subsystem" => "remoting"] - Current version: 2.0.0; legacy
version: 1.2.0 =======
--- Problems for relative address to root []:
Missing child types in current: []; missing in legacy [http-connector]
--- Problems for relative address to root ["remote-outbound-connection" => "*"]:
Missing attributes in current: []; missing in legacy [protocol]
Missing parameters for operation 'add' in current: []; missing in legacy [protocol]
-- SNIP --
====== Resource root address: ["subsystem" => "weld"] - Current version: 2.0.0; legacy version:
1.0.0 =======
--- Problems for relative address to root []:
Missing attributes in current: []; missing in legacy [require-bean-descriptor,
non-portable-mode]
Missing parameters for operation 'add' in current: []; missing in legacy
[require-bean-descriptor, non-portable-mode]

Done comparison of STANDALONE!
```

So we can see that for the `remoting` subsystem, we have added a child type called `http-connector`, and we have added an attribute called `protocol` (they are missing in legacy).
in the `weld` subsystem, we have added the `require-bean-descriptor` and `non-portable-mode` attributes in the current version. It will also point out other issues like changed attribute types, changed defaults etc.

Warning

Note that CompareModelVersionsUtil simply inspects the raw resource descriptions of the specified legacy and current models. Its results show the differences between the two. They do not take into account whether one or more transformers have already been written for those versions differences. You will need to check that transformers are not already in place for those versions.

One final point to consider are that some subsystems register runtime-only resources and operations. For example the `modcluster` subsystem has a `stop` method. These do not get registered on the `DC`, e.g. there is no `/profile=full-ha/subsystem=modcluster:stop` operation, it only exists on the servers, for example `/host=xxx/server=server-one/subsystem=modcluster:stop`. What this means is that you don't have to transform such operations and resources. The reason is they are not callable on the DC, and so do not need propagation to the servers in the domain, which in turn means no transformation is needed.

11.5.6 How do I write a transformer?

There are two APIs available to write transformers for a resource. There is the original low-level API where you register transformers directly, the general idea is that you get hold of a `TransformersSubRegistration` for each level and implement the `ResourceTransformer`, `OperationTransformer` and `PathAddressTransformer` interfaces directly. It is, however, a pretty complex thing to do, so we recommend the other approach. For completeness here is the entry point to handling transformation in this way.

```
public class SomeExtension implements Extension {

    private static final String SUBSYSTEM_NAME = "my-subsystem"'

    private static final int MANAGEMENT_API_MAJOR_VERSION = 2;
    private static final int MANAGEMENT_API_MINOR_VERSION = 0;
    private static final int MANAGEMENT_API_MICRO_VERSION = 0;

    @Override
    public void initialize(ExtensionContext context) {
        SubsystemRegistration registration = context.registerSubsystem(SUBSYSTEM_NAME,
MANAGEMENT_API_MAJOR_VERSION,
                MANAGEMENT_API_MINOR_VERSION, MANAGEMENT_API_MICRO_VERSION);
        //Register the resource definitions
        ....
    }

    static void registerTransformers(final SubsystemRegistration subsystem) {
        registerTransformers_1_1_0(subsystem);
        registerTransformers_1_2_0(subsystem);
    }

    /**
     * Registers transformers from the current version to ModelVersion 1.1.0
     */
    private static void registerTransformers_1_1_0(final SubsystemRegistration subsystem) {
        final ModelVersion version = ModelVersion.create(1, 1, 0);

        //The default resource transformer forwards all operations
        final TransformersSubRegistration registration =
subsystem.registerModelTransformers(version, ResourceTransformer.DEFAULT);
        final TransformersSubRegistration child =
registration.registerSubResource(PathElement.pathElement("child"));
        //We can do more things on the TransformersSubRegistation instances


        registerRelayTransformers(stack);
    }
```

Having implemented a number of transformers using the above approach, we decided to simplify things, so we introduced the `org.jboss.as.controller.transform.description.ResourceTransformationDescriptionB` API. It is a lot simpler and avoids a lot of the duplication of functionality required by the low-level API approach. While it doesn't give you the full power that the low-level API does, we found that there are very few places in the WildFly codebase where this does not work, so we will focus on the `ResourceTransformationDescriptionBuilder` API here. (If you come across a problem where this does not work, get in touch with someone from the WildFly Domain Management Team and we should be able to help). The builder API makes all the nasty calls to `TransformersSubRegistration` for you under the hood. It also allows you to fall back to the low-level API in places, although that will not be covered in the current version of this guide. The entry point for using the builder API here is taken from the WeldExtension (in current WildFly this has ModelVersion 2.0.0).

```
private void registerTransformers(SubsystemRegistration subsystem) {
        ResourceTransformationDescriptionBuilder builder =
TransformationDescriptionBuilder.Factory.createSubsystemInstance();
        //These new attributes are assumed to be 'true' in the old version but default to false
in the current version. So discard if 'true' and reject if 'undefined'.
        builder.getAttributeBuilder()
                .setDiscard(new DiscardAttributeChecker.DiscardAttributeValueChecker(false,
false, new ModelNode(true)),
                        WeldResourceDefinition.NON_PORTABLE_MODE_ATTRIBUTE,
WeldResourceDefinition.REQUIRE_BEAN_DESCRIPTOR_ATTRIBUTE)
                .addRejectCheck(new RejectAttributeChecker.DefaultRejectAttributeChecker() {

                    @Override
                    public String getRejectionLogMessage(Map<String, ModelNode> attributes) {
                        return
WeldMessages.MESSAGES.rejectAttributesMustBeTrue(attributes.keySet());
                    }

                    @Override
                    protected boolean rejectAttribute(PathAddress address, String attributeName,
ModelNode attributeValue,
                            TransformationContext context) {
                        //This will not get called if it was discarded, so reject if it is
undefined (default==false) or if defined and != 'true'
                        return !attributeValue.isDefined() ||
!attributeValue.asString().equals("true");
                    }
                }, WeldResourceDefinition.NON_PORTABLE_MODE_ATTRIBUTE,
WeldResourceDefinition.REQUIRE_BEAN_DESCRIPTOR_ATTRIBUTE)
                .end();
        TransformationDescription.Tools.register(builder.build(), subsystem,
ModelVersion.create(1, 0, 0));
    }
```

Here we register a `discard check` and a `reject check`. As mentioned in Attribute transformation lifecycle all attributes are inspected for whether they should be discarded first. Then all attributes which were not discarded are checked for if they should be rejected. We will dig more into what this code means in the next few sections, but in short it means that we discard the `require-bean-descriptor` and `non-portable` attributes on the `weld` subsystem resource if they have the value `true`. If they have any other value, they will not get discarded and so reach the reject check, which will reject the transformation of the attributes if they have any other value.

Here we are saying that we should discard the `require-bean-descriptor` and `non-portable-mode` attributes on the `weld` subsystem resource if they are undefined, and reject them if they are defined. So that means that if the weld subsystem looks like

```
{
        "non-portable-mode" => false,
        "require-bean-descriptor" => false
    }
```

or

```
{
        "non-portable-mode" => undefined,
        "require-bean-descriptor" => undefined
    }
```

or any other combination (the default values for these attributes if undefined is `false`) we will reject the transformation for the slave legacy HC.

If the resource has true for these attributes:

```
{
        "non-portable-mode" => true,
        "require-bean-descriptor" => true
    }
```

they both get discarded (i.e. removed), so they will not get inspected for rejection, and an empty model not containing these attributes gets sent to the legacy HC.

Here we will discuss this API a bit more, to outline the most important features/most commonly needed tasks.

ResourceTransformationDescriptionBuilder

The `ResourceTransformationDescriptionBuilder` contains transformations for a resource type. The initial one is for the subsystem, obtained by the following call:

```
ResourceTransformationDescriptionBuilder subsystemBuilder =
TransformationDescriptionBuilder.Factory.createSubsystemInstance();
```

The `ResourceTransformationDescriptionBuilder` contains functionality for how to handle child resources, which we will look at in this section. It is also the entry point to how to handle transformation of attributes as we will see in AttributeTransformationDescriptionBuilder. Also, it allows you to further override operation transformation as discussed in OperationTransformationOverrideBuilder. When we have finished with our builder, we register it with the `SubsystemRegistration` against the target ModelVersion.

```
TransformationDescription.Tools.register(subsystemBuilder.build(), subsystem,
ModelVersion.create(1, 0, 0));
```

Important

If you have several old ModelVersions you could be transforming to, you need a separate builder for each of those.

Silently discard child resources

To make the `ResourceTransformationDescriptionBuilder` do something, we need to call some of its methods. For example, if we want to silently discard a child resource, we can do

```
subsystemBuilder.discardChildResource(PathElement.pathElement("child", "discarded"));
```

This means that any usage of `/subsystem=my-subsystem/child=discarded` never make it to the legacy slave HC running ModelVersion 1.0.0. During the initial domain model transfer, that part of the serialized domain model is stripped out, and any operations on this address are not forwarded on to the legacy slave HCs running that version of the subsystem. (For brevity this section will leave out the leading `/profile=xxx` part used in domain mode, and use `/subsystem=my-subsystem` as the 'top-level' address).

Warning

Note that discarding, although the simplest option in theory, is **rarely the right thing to do**.

The presence of the defined child normally implies some behaviour on the DC, and that behaviour is not available on the legacy slave HC, so normally rejection is a better policy for those cases. Remember we can have different profiles targeting different groups of versions of legacy slave HCs.

Reject child resource

If we want to reject transformation if a child resource exists, we can do

```
subsystemBuilder.rejectChildResource(PathElement.pathElement("child", "reject"));
```

Now, if there are any legacy slaves running ModelVersion 1.0.0, any usage of `/subsystem=my-subsystem/child=reject` will get rejected for those slaves. Both during the initial domain model transfer, and if any operations are invoked on that address. For example the `remoting` subsystem did not have a `http-connector=*` child until ModelVersion 2.0.0, so it is set up to reject that child when transforming to legacy HCs for all previous ModelVersions (1.1.0, 1.2.0 and 1.3.0). (See Rejection in resource transformers and Rejection in operation transformers for exactly what happens when something is rejected).

Redirect address for child resource

Sometimes we rename the addresses for a child resource between model versions. To do that we use one of the `addChildRedirection()` methods, note that these also return a builder for the child resource (since we are not rejecting or discarding it), we can do this for all children of a given type:

```
ResourceTransformationDescriptionBuilder childBuilder =
       subsystemBuilder.addChildRedirection(PathElement.pathElement("newChild"),
PathElement.pathElement("oldChild");
```

Now, in the initial domain transfer `/subsystem=my-subsystem/newChild=test` becomes `/subsystem=my-subsystem/oldChild=test`. Similarly all operations against the former address get mapped to the latter when executing operations on the DC before sending them to the legacy slave HC running ModelVersion 1.1.0 of the subsystem.

We can also rename a specific named child:

```
ResourceTransformationDescriptionBuilder childBuilder =
       subsystemBuilder.addChildRedirection(PathElement.pathElement("newChild", "newName"),
PathElement.pathElement("oldChild", "oldName");
```

Now, `/subsystem=my-subsystem/newChild=newName` becomes `/subsystem=my-subsystem/oldChild=oldName` both in the initial domain transfer, and when mapping operations to the legacy slave. For example, under the web subsystem `ssl=configuration` got renamed to `configuration=ssl` in later versions, meaning we need a redirect from `configuration=ssl` to `ssl=configuration` in its transformers.

Getting a child resource builder

Sometimes we don't want to transform the subsystem resource, but we want to transform something in one of its child resources. Again, since we are not discarding or rejecting, we get a reference to the builder for the child resource.

```
ResourceTransformationDescriptionBuilder childBuilder =
       subsystemBuilder.addChildResource(PathElement.pathElement("some-child"));
    //We don't actually want to transform anything in /subsystem-my-subsystem/some-child=*
either :-)
    //We are interested in /subsystem-my-subsystem/some-child=*/another-level
    ResourceTransformationDescriptionBuilder anotherBuilder =
       childBuilder.addChildResource(PathElement.pathElement("another-level"));

    //Use anotherBuilder to add child-resource and/or attribute transformation
    ....
```

AttributeTransformationDescriptionBuilder

To transform attributes you call `ResourceTransformationDescriptionBuilder.getAttributeBuilder()` which returns you a `AttributeTransformationDescriptionBuilder` which is used to define transformation for the resource's attributes. For example this gets the attribute builder for the subsystem resource:

```
AttributeTransformationDescriptionBuilder attributeBuilder =
subSystemBuilder.getAttributeBuilder();
```

or we could get it for one of the child resources:

```
ResourceTransformationDescriptionBuilder childBuilder =
       subsystemBuilder.addChildResource(PathElement.pathElement("some-child"));
    AttributeTransformationDescriptionBuilder attributeBuilder =
childBuilder.getAttributeBuilder();
```

The attribute transformations defined by the `AttributeTransformationDescriptionBuilder` will also impact the parameters to all operations defined on the resource. This means that if you have defined the `example` attribute of `/subsystem=my-subsystem/some-child=*` to reject transformation if its value is `true`, the inital domain transfer will reject if it is `true`, also the transformation of the following operations will reject:

```
/subsystem=my-subsystem/some-child=test:add(example=true)
    /subsystem=my-subsystem:write-attribute(name=example, value=true)
    /subsystem=my-subsystem:custom-operation(example=true)
```

The following operations will pass in this example, since the `example` attribute is not getting set to `true`

```
/subsystem=my-subsystem/some-child=test:add(example=false)
    /subsystem=my-subsystem/some-child=test:add()             //Here it 'example' is simply left
undefined
    /subsystem=my-subsystem:write-attribute(name=example, value=false)
    /subsystem=my-subsystem:undefine-attribute(name=example)  //Again this makes 'example'
undefined
    /subsystem=my-subsystem:custom-operation(example=false)
```

For the rest of the examples in this section we assume that the `attributeBuilder` is for `/subsystem=my-subsystem`

Attribute transformation lifecycle

There is a well defined lifecycle used for attribute transformation that is worth explaining before jumping into specifics. Transformation is done in the following phases, in the following order:

1. `discard` - All attributes in the domain model transfer or invoked operation that have been registered for a discard check, are checked to see if the attribute should be discarded. If an attribute should be discarded, it is removed from the resource's attributes/operation's parameters and it does not get passed to the next phases. Once discarded it does not get sent to the legacy slave HC.
2. `reject` - All attributes that have been registered for a reject check (and which not have been discarded) are checked to see if the attribute should be rejected. As explained in Rejection in resource transformers and Rejection in operation transformers exactly what happens when something is rejected varies depending on whether we are transforming a resource or an operation, and the version of the legacy slave HC we are transforming for. If a transformer rejects an attribute, all other reject transformers still get invoked, and the next phases also get invoked. This is because we don't know in all cases what will happen if a reject happens. Although this might sound cumbersome, in practice it actually makes it easier to write transformers since you only need one kind regardless of if it is a resource, an operation, and legacy slave HC version. However, as we will see in Common transformation use-cases, it means some extra checks are needed when writing reject and convert transformers.
3. `convert` - All attributes that have been registered for conversion are checked to see if the attribute should be converted. If the attribute does not exist in the original operation/resource it may be introduced. This is useful for setting default values for the target legacy slave HC.
4. `rename` - All attributes registered for renaming are renamed.

Next, let us have a look at how to register attributes for each of these phases.

Discarding attributes

The general idea behind a discard is that we remove attributes which do not exist in the legacy slave HC's model. However, as hopefully described below, we normally can't simply discard everything, we need to check the values first.

To discard an attribute we need an instance of `org.jboss.as.controller.transform.description.DiscardAttributeChecker`, and call the following method on the `AttributeTransformationDescriptionBuilder`:

```
DiscardAttributeChecker discardCheckerA = ....;
     attributeBuilder.setDiscard(discardCheckerA, "attr1", "attr2");
```

As shown, you can register the `DiscardAttributeChecker` for several attributes at once, in the above example both `attr1` and `attr2` get checked for if they should be discarded. You can also register different `DiscardAttributeChecker` instances for different attributes:

```
DiscardAttributeChecker discardCheckerA = ....;
     DiscardAttributeChecker discardCheckerB = ....;
     attributeBuilder.setDiscard(discardCheckerA, "attr1");
     attributeBuilder.setDiscard(discardCheckerA, "attr2");
```

Note that you can only have one `DiscardAttributeChecker` per attribute, so the following would cause an error (if running with assertions enabled, otherwise `discardCheckerB` will overwrite `discardCheckerA`):

```
DiscardAttributeChecker discardCheckerA = ....;
     DiscardAttributeChecker discardCheckerB = ....;
     attributeBuilder.setDiscard(discardCheckerA, "attr1");
     attributeBuilder.setDiscard(discardCheckerB, "attr1");
```

The DiscardAttributeChecker interface

`org.jboss.as.controller.transform.description.DiscardAttributeChecker` contains both the `DiscardAttributeChecker` and some helper implementations. The implementations of this interface get called for each attribute they are registered against. The interface itself is quite simple:

```
public interface DiscardAttributeChecker {

    /**
     * Returns {@code true} if the attribute should be discarded if expressions are used
     *
     * @return whether to discard if expressions are used
     */
    boolean isDiscardExpressions();
```

Return `true` here to discard the attribute if it is an expression. If it is an expression, and this method returns `true`, the `isOperationParameterDiscardable` and `isResourceAttributeDiscardable` methods will not get called.

```
/**
     * Returns {@code true} if the attribute should be discarded if it is undefined
     *
     * @return whether to discard if the attribute is undefined
     */
    boolean isDiscardUndefined();
```

Return `true` here to discard the attribute if it is `undefined`. If it is `undefined`, and this method returns `true`, the `isDiscardExpressions`, `isOperationParameterDiscardable` and `isResourceAttributeDiscardable` methods will not get called.

```
/**
     * Gets whether the given operation parameter can be discarded
     *
     * @param address the address of the operation
     * @param attributeName the name of the operation parameter.
     * @param attributeValue the value of the operation parameter.
     * @param operation the operation executed. This is unmodifiable.
     * @param context the context of the transformation
     *
     * @return {@code true} if the operation parameter value should be discarded, {@code false}
otherwise.
     */
    boolean isOperationParameterDiscardable(PathAddress address, String attributeName, ModelNode
attributeValue, ModelNode operation, TransformationContext context);
```

If we are transforming an operation, this method gets called for each operation parameter. We have access to the address of the operation, the name and value of the operation parameter, an unmodifiable copy of the original operation and the `TransformationContext`. The `TransformationContext` allows you access to the original resource the operation is working on before any transformation happened, which is useful if you want to check other values in the resource if this is, say a `write-attribute` operation. Return `true` to discard the operation.

```
/**
     * Gets whether the given attribute can be discarded
     *
     * @param address the address of the resource
     * @param attributeName the name of the attribute
     * @param attributeValue the value of the attribute
     * @param context the context of the transformation
     *
     * @return {@code true} if the attribute value should be discarded, {@code false} otherwise.
     */
    boolean isResourceAttributeDiscardable(PathAddress address, String attributeName, ModelNode
attributeValue, TransformationContext context);
```

If we are transforming a resource, this method gets called for each attribute in the resource. We have access to the address of the resource, the name and value of the attribute, and the `TransformationContext`. Return `true` to discard the operation.

```
}
```

DiscardAttributeChecker helper classes/implementations

`DiscardAttributeChecker` contains a few helper implementations for the most common cases to save you writing the same stuff again and again.

DiscardAttributeChecker.DefaultDiscardAttributeChecker

`DiscardAttributeChecker.DefaultDiscardAttributeChecker` is an abstract convenience class. In most cases you don't need a separate check for if an operation or a resource is being transformed, so it makes both the `isResourceAttributeDiscardable()` and `isOperationParameterDiscardable()` methods call the following method.

```
protected abstract boolean isValueDiscardable(PathAddress address, String attributeName,
ModelNode attributeValue, TransformationContext context);
```

All you lose, in the case of an operation transformation, is the name of the transformed operation. The constructor of `DiscardAttributeChecker.DefaultDiscardAttributeChecker` also allows you to define values for `isDiscardExpressions()` and `isDiscardUndefined()`.

DiscardAttributeChecker.DiscardAttributeValueChecker

This is another convenience class, which allows you to discard an attribute if it has one or more values. Here is a real-world example from the `jpa` subsystem:

```
private void initializeTransformers_1_1_0(SubsystemRegistration subsystemRegistration) {
        ResourceTransformationDescriptionBuilder builder =
TransformationDescriptionBuilder.Factory.createSubsystemInstance();
        builder.getAttributeBuilder()
            .setDiscard(
                  new DiscardAttributeChecker.DiscardAttributeValueChecker(new
ModelNode(ExtendedPersistenceInheritance.DEEP.toString())),
                  JPADefinition.DEFAULT_EXTENDEDPERSISTENCE_INHERITANCE)
            .addRejectCheck(RejectAttributeChecker.DEFINED,
JPADefinition.DEFAULT_EXTENDEDPERSISTENCE_INHERITANCE)
            .end();
        TransformationDescription.Tools.register(builder.build(), subsystemRegistration,
ModelVersion.create(1, 1, 0));
    }
```

We will come back to the reject checks in the Rejecting attributes section. We are saying that we should discard the `JPADefinition.DEFAULT_EXTENDEDPERSISTENCE_INHERITANCE` attribute if it has the value `deep`. The reasoning here is that this attribute did not exist in the old model, but the legacy slave HCs *implied behaviour* is that this was `deep`. In the current version we added the possibility to toggle this setting, but only `deep` is consistent with what is available in the legacy slave HC. In this case we are using the constructor for `DiscardAttributeChecker.DiscardAttributeValueChecker` which says don't discard if it uses expressions, and discard if it is `undefined`. If it is `undefined` in the current model, looking at the default value of `JPADefinition.DEFAULT_EXTENDEDPERSISTENCE_INHERITANCE`, it is `deep`, so a discard is in line with the implied legacy behaviour. If an expression is used, we cannot discard since we have no idea what the expression will resolve to on the slave HC.

DiscardAttributeChecker.ALWAYS

`DiscardAttributeChecker.ALWAYS` will always discard an attribute. Use this sparingly, since normally the presence of an attribute in the current model implies some behaviour should be turned on, and if that does not exist in the legacy model it implies that that behaviour does not exist in the legacy slave HC and its servers. Normally the legacy slave HC's subsystem has some implied behaviour which is better checked for by using a `DiscardAttributeChecker.DiscardAttributeValueChecker`. One valid use for `DiscardAttributeChecker.ALWAYS` can be found in the `ejb3` subsystem:

```
private static void registerTransformers_1_1_0(SubsystemRegistration subsystemRegistration) {
        ResourceTransformationDescriptionBuilder builder =
TransformationDescriptionBuilder.Factory.createSubsystemInstance()
                .getAttributeBuilder()
                 ...
                // We can always discard this attribute, because it's meaningless without the
security-manager subsystem, and
                // a legacy slave can't have that subsystem in its profile.
                .setDiscard(DiscardAttributeChecker.ALWAYS,
EJB3SubsystemRootResourceDefinition.DISABLE_DEFAULT_EJB_PERMISSIONS)
   ...
```

As the comment says, this attribute only makes sense with the security-manager susbsystem, which does not exist on legacy slaves running ModelVersion 1.1.0 of the `ejb3` subsystem.

DiscardAttributeChecker.UNDEFINED

`DiscardAttributeChecker.UNDEFINED` will discard an attribute if it is `undefined`. This is normally safer than `DiscardAttributeChecker.ALWAYS` since the attribute is not set in the current model, we don't need to send it to the legacy model. However, you should check that this attribute not existing in the legacy slave HC, implies the same functionality as being undefined in the current DC.

Rejecting attributes

The next step is to check attributes and values which we know for sure will not work on the target legacy slave HC.

To reject an attribute we need an instance of `org.jboss.as.controller.transform.description.RejectAttributeChecker`, and call the following method on the `AttributeTransformationDescriptionBuilder`:

```
RejectAttributeChecker rejectCheckerA = ....;
     attributeBuilder.addRejectCheck(rejectCheckerA, "attr1", "attr2");
```

As shown you can register the `RejectAttributeChecker` for several attributes at once, in the above example both `attr1` and `attr2` get checked for if they should be discarded. You can also register different `RejectAttributeChecker` instances for different attributes:

```
RejectAttributeChecker rejectCheckerA = ....;
     RejectAttributeChecker rejectCheckerB = ....;
     attributeBuilder.addRejectCheck(rejectCheckerA, "attr1");
     attributeBuilder.addRejectCheck(rejectCheckerB, "attr2");
```

You can also register several `RejectAttributeChecker` instances per attribute

```
RejectAttributeChecker rejectCheckerA = ....;
     RejectAttributeChecker rejectCheckerB = ....;
     attributeBuilder.addRejectCheck(rejectCheckerA, "attr1");
     attributeBuilder.addRejectCheck(rejectCheckerB, "attr1, "attr2");
```

In this case `attr1` gets both `rejectCheckerA` and `rejectCheckerB`. For attributes with several `RejectAttributeChecker` registered, they get processed in the order that they have been added. So when checking `attr1` for rejection, `rejectCheckerA` gets run before `rejectCheckerB`. As mentioned in Attribute transformation lifecycle, if an attribute is rejected, we still invoke the rest of the reject checkers.

The RejectAttributeChecker interface

`org.jboss.as.controller.transform.description.RejectAttributeChecker` contains both the `RejectAttributeChecker` and some helper implementations. The implementations of this interface get called for each attribute they are registered against. The interface itself is quite simple, and its main methods are similar to `DiscardAttributeChecker`:

```
public interface RejectAttributeChecker {
    /**
     * Determines whether the given operation parameter value is not understandable by the
target process and needs
     * to be rejected.
     *
     * @param address        the address of the operation
     * @param attributeName  the name of the attribute
     * @param attributeValue the value of the attribute
     * @param operation      the operation executed. This is unmodifiable.
     * @param context        the context of the transformation
     * @return {@code true} if the parameter value is not understandable by the target process
and so needs to be rejected, {@code false} otherwise.
     */
    boolean rejectOperationParameter(PathAddress address, String attributeName, ModelNode
attributeValue, ModelNode operation, TransformationContext context);
```

If we are transforming an operation, this method gets called for each operation parameter. We have access to the address of the operation, the name and value of the operation parameter, an unmodifiable copy of the original operation and the `TransformationContext`. The `TransformationContext` allows you access to the original resource the operation is working on before any transformation happened, which is useful if you want to check other values in the resource if this is, say a `write-attribute` operation. Return `true` to reject the operation.

```
/**
     * Gets whether the given resource attribute value is not understandable by the target
process and needs
     * to be rejected.
     *
     * @param address        the address of the resource
     * @param attributeName  the name of the attribute
     * @param attributeValue the value of the attribute
     * @param context        the context of the transformation
     * @return {@code true} if the attribute value is not understandable by the target process
and so needs to be rejected, {@code false} otherwise.
     */
    boolean rejectResourceAttribute(PathAddress address, String attributeName, ModelNode
attributeValue, TransformationContext context);
```

If we are transforming a resource, this method gets called for each attribute in the resource. We have access to the address of the resource, the name and value of the attribute, and the `TransformationContext`. Return `true` to discard the operation.

```
/**
     * Returns the log message id used by this checker. This is used to group it so that all
attributes failing a type of rejection
     * end up in the same error message
     *
     * @return the log message id
     */
    String getRejectionLogMessageId();
```

Here we need a unique id for the log message from the `RejectAttributeChecker`. It is used to group rejected attributes by their log message. A typical implementation will contain {{return getRejectionLogMessage(Collections.<String, ModelNode>emptyMap());}

```
/**
     * Gets the log message if the attribute failed rejection
     *
     * @param attributes a map of all attributes failed in this checker and their values
     * @return the formatted log message
     */
    String getRejectionLogMessage(Map<String, ModelNode> attributes);
```

Here we return a message saying why the attributes were rejected, with the possibility to format the message to include the names of all the rejected attributes and the values they had.

```
}
```

RejectAttributeChecker helper classes/implementations

`RejectAttributeChecker` contains a few helper classes for the most common scenarios to save you from writing the same stuff again and again.

RejectAttributeChecker.DefaultRejectAttributeChecker

`RejectAttributeChecker.DefaultRejectAttributeChecker` is an abstract convenience class. In most cases you don't need a separate check for if an operation or a resource is being transformed, so it makes both the `rejectOperationParameter()` and `rejectResourceAttribute()` methods call the following method.

```
protected abstract boolean rejectAttribute(PathAddress address, String attributeName, ModelNode
attributeValue, TransformationContext context);
```

Like `DefaultDiscardAttributeChecker`, all you loose is the name of the transformed operation, in the case of operation transformation.

RejectAttributeChecker.DEFINED

`RejectAttributeChecker.DEFINED` is used to reject any attribute that has a defined value. Normally this is because the attribute does not exist on the target legacy slave HC. A typical use case for these is for the *implied behavior* example we looked at in the `jpa` subsystem in DiscardAttributeChecker.DiscardAttributeValueChecker

```
private void initializeTransformers_1_1_0(SubsystemRegistration subsystemRegistration) {
        ResourceTransformationDescriptionBuilder builder =
TransformationDescriptionBuilder.Factory.createSubsystemInstance();
        builder.getAttributeBuilder()
            .setDiscard(
                  new DiscardAttributeChecker.DiscardAttributeValueChecker(new
ModelNode(ExtendedPersistenceInheritance.DEEP.toString())),
                  JPADefinition.DEFAULT_EXTENDEDPERSISTENCE_INHERITANCE)
            .addRejectCheck(RejectAttributeChecker.DEFINED,
JPADefinition.DEFAULT_EXTENDEDPERSISTENCE_INHERITANCE)
            .end();
        TransformationDescription.Tools.register(builder.build(), subsystemRegistration,
ModelVersion.create(1, 1, 0));
    }
```

So we discard the `JPADefinition.DEFAULT_EXTENDEDPERSISTENCE_INHERITANCE` value if it is not an expression, and also has the value `deep`. Now if it was not discarded, it would will still be defined so we reject it.

Important

Reject and discard often work in pairs.

RejectAttributeChecker.SIMPLE_EXPRESSIONS

`RejectAttributeChecker.SIMPLE_EXPRESSIONS` can be used to reject an attribute that contains expressions. This was used a lot for transformations to subsystems in JBoss AS 7.1.x, since we had not fully realized the importance of where to support expressions until JBoss AS 7.2.0 was released, so a lot of attributes in earlier versions were missing expressions support.

RejectAttributeChecker.ListRejectAttributeChecker

The `RejectAttributeChecker}}s we have seen so far work on simple attributes, i.e. where the attribute has a ModelType which is one of the primitives. We also have a {{RejectAttributeChecker.ListRejectAttributeChecker` which allows you to define a checker for the elements of a list, when the type of an attribute is `ModelType.LIST`.

```
attributeBuilder
            .addRejectCheck(new ListRejectAttributeChecker(RejectAttributeChecker.EXPRESSIONS),
"attr1");
```

For `attr1` it will check each element of the list and run `RejectAttributeChecker.EXPRESSIONS` to check that each element is not an expression. You can of course pass in another kind of `RejectAttributeChecker` to check the elements as well.

RejectAttributeChecker.ObjectFieldsRejectAttributeChecker

For attributes where the type is `ModelType.OBJECT` we have `RejectAttributeChecker.ObjectFieldsRejectAttributeChecker` which allows you to register different reject checkers for the different fields of the registered object.

```
Map<String, RejectAttributeChecker> fieldRejectCheckers = new HashMap<String,
RejectAttributeChecker>();
    fieldRejectCheckers.put("time", RejectAttributeChecker.SIMPLE_EXPRESSIONS);
    fieldRejectCheckers.put("unit", "Lunar Month");
    attributeBuilder
            .addRejectCheck(new ObjectFieldsRejectAttributeChecker(fieldRejectCheckers),
"attr1");
```

Now if `attr1` is a complex type where `attr1.get("time").getType() == ModelType.EXPRESSION` or `attr1.get("unit").asString().equals("Lunar Month")` we reject the attribute.

Converting attributes

To convert an attribute you register an `org.jboss.as.controller.transform.description.AttributeConverter` instance against the attributes you want to convert:

```
AttributeConverter converterA = ...;
    AttributeConverter converterB = ...;
    attributeBuilder
            .setValueConverter(converterA, "attr1", "attr2");
    attributeBuilder
            .setValueConverter(converterB, "attr3");
```

Now if `attr1` and `attr2` get converted with `converterA`, while `attr3` gets converted with `converterB`.

The AttributeConverter interface

The `AttributeConverter` interface gets called for each attribute for which the `AttributeConverter` has been registered

```
public interface AttributeConverter {

    /**
     * Converts an operation parameter
     *
     * @param address the address of the operation
     * @param attributeName the name of the operation parameter
     * @param attributeValue the value of the operation parameter to be converted
     * @param operation the operation executed. This is unmodifiable.
     * @param context the context of the transformation
     */
    void convertOperationParameter(PathAddress address, String attributeName, ModelNode
attributeValue, ModelNode operation, TransformationContext context);
```

If we are transforming an operation, this method gets called for each operation parameter for which the con. We have access to the address of the operation, the name and value of the operation parameter, an unmodifiable copy of the original operation and the `TransformationContext`. The `TransformationContext` allows you access to the original resource the operation is working on before any transformation happened, which is useful if you want to check other values in the resource if this is, say a write-attribute operation. To change the attribute value, you modify the `attributeValue`.

```
/**
     * Converts a resource attribute
     *
     * @param address the address of the operation
     * @param attributeName the name of the attribute
     * @param attributeValue the value of the attribute to be converted
     * @param context the context of the transformation
     */
    void convertResourceAttribute(PathAddress address, String attributeName, ModelNode
attributeValue, TransformationContext context);
```

If we are transforming a resource, this method gets called for each attribute in the resource. We have access to the address of the resource, the name and value of the attribute, and the `TransformationContext`. To change the attribute value, you modify the `attributeValue`.

```
}
```

A hypothetical example is if the current and legacy subsystems both contain an attribute called `timeout`. In the legacy model this was specified to be milliseconds, however in the current model it has been changed to be seconds, hence we need to convert the value when sending it to slave HCs using the legacy model:

```
AttributeConverter secondsToMs = new AttributeConverter.DefaultAttributeConverter() {
                  @Override
                  protected void convertAttribute(PathAddress address, String attributeName,
ModelNode attributeValue,
                           TransformationContext context) {
                      if (attributeValue.isDefined()) {
                           int seconds = attributeValue.asInt();
                           int milliseconds = seconds * 1000;
                           attributeValue.set(milliseconds);
                      }
                  }
          };

     attributeBuilder.
          .setValueConverter(secondsToMs , "timeout")
```

We need to be a bit careful here. If the `timeout` attribute is an expression our nice conversion will not work, so we need to add a reject check to make sure it is not an expression as well:

```
attributeBuilder.
          .addRejectCheck(SIMPLE_EXPRESSIONS, "timeout")
          .setValueConverter(secondsToMs , "timeout")
```

Now it should be fine.

`AttributeConverter.DefaultAttributeConverter` is is an abstract convenience class. In most cases you don't need a separate check for if an operation or a resource is being transformed, so it makes both the convertOperationParameter() and convertResourceAttribute() methods call the following method.

```
protected abstract void convertAttribute(PathAddress address, String attributeName, ModelNode
attributeValue, TransformationContext context);
```

Like `DefaultDiscardAttributeChecker` and `DefaultRejectAttributeChecker`, all you loose is the name of the transformed operation, in the case of operation transformation.

Introducing attributes during transformation
Say both the current and the legacy models have an attribute called `port`. In the legacy version this attribute had to be specified, and the default xml configuration had `1234` for its value. In the current version this attribute has been made optional with a default value of `1234` so that it does not need to be specified. When transforming to a slave HC using the old version we will need to introduce this attribute if the new model does not contain it:

```
attributeBuilder.
         setValueConverter(AttributeConverter.Factory.createHardCoded(new ModelNode(1234) true),
"port");
```

So what this factory method does is to create an implementation of `AttributeConverter.DefaultAttributeConverter` where in `convertAttribute()` we set `attributeValue` to have the value `1234` if it is `undefined`. As long as `attributeValue` gets set in that method it will get set in the model, regardless of if it existed already or not.

Renaming attributes

To rename an attribute, you simply do

```
attributeBuilder.addRename("my-name", "legacy-name");
```

Now, in the initial domain transfer to the legacy slave HC, we rename `/subsystem=my-subsystem`'s `my-name` attribute to `legacy-name`. Also, the operations involving this attribute are affected, so

```
/subsystem=my-subsystem/:add(my-name=true)  ->
         /subsystem=my-subsystem/:add(legacy-name=true)
    /subsystem=my-subsystem:write-attribute(name=my-name, value=true) ->
         /subsystem=my-subsystem:write-attribute(name=legacy-name, value=true)
    /subsystem=my-subsystem:undefine-attribute(name=my-name) ->
         /subsystem=my-subsystem:undefine-attribute(name=legacy-name)
```

OperationTransformationOverrideBuilder

All operations on a resource automatically get the same transformations on their parameters as set up by the `AttributeTransformationDescriptionBuilder`. In some cases you might want to change this, so you can use the `OperationTransformationOverrideBuilder`, which is got from:

```
OperationTransformationOverrideBuilder operationBuilder =
subSystemBuilder.addOperationTransformationOverride("some-operation");
```

In this case the operation will now no longer inherit the attribute/operation parameter transformations, so they are effectively turned off. In other cases you might want to include them by calling `inheritResourceAttributeDefinitions()`, and to include some more checks (the `OperationTransformationBuilder` interface has all the methods found in `AttributeTransformationBuilder`:

```
OperationTransformationOverrideBuilder operationBuilder =
subSystemBuilder.addOperationTransformationOverride("some-operation");
    operationBuilder.inheritResourceAttributeDefinitions();
    operationBuilder.setValueConverter(AttributeConverter.Factory.createHardCoded(new
ModelNode(1234) true), "port");
```

You can also rename operations, in this case the operation `some-operation` gets renamed to `legacy-operation` before getting sent to the legacy slave HC.

```
OperationTransformationOverrideBuilder operationBuilder =
subSystemBuilder.addOperationTransformationOverride("some-operation");
    operationBuilder.rename("legacy-operation");
```

11.5.7 Evolving transformers with subsystem ModelVersions

Say you have a subsystem with ModelVersions 1.0.0 and 1.1.0. There will (hopefully!) already be transformers in place for 1.1.0 to 1.0.0 transformations. Let's say that the transformers registration looks like:

```
public class SomeExtension implements Extension {

    private static final String SUBSYSTEM_NAME = "my-subsystem"'

    private static final int MANAGEMENT_API_MAJOR_VERSION = 1;
    private static final int MANAGEMENT_API_MINOR_VERSION = 1;
    private static final int MANAGEMENT_API_MICRO_VERSION = 0;

    @Override
    public void initialize(ExtensionContext context) {
        SubsystemRegistration registration = context.registerSubsystem(SUBSYSTEM_NAME,
MANAGEMENT_API_MAJOR_VERSION,
                MANAGEMENT_API_MINOR_VERSION, MANAGEMENT_API_MICRO_VERSION);
        //Register the resource definitions
        ....
    }

    private void registerTransformers(final SubsystemRegistration subsystem) {
        registerTransformers_1_0_0(subsystem);
    }

    /**
     * Registers transformers from the current version to ModelVersion 1.0.0
     */
    private void registerTransformers_1_0_0(SubsystemRegistration subsystem) {
        ResourceTransformationDescriptionBuilder builder =
TransformationDescriptionBuilder.Factory.createSubsystemInstance();
        builder.getAttributeBuilder()
            .addRejectCheck(RejectAttributeChecker.DEFINED, "attr1")
            .end();
        TransformationDescription.Tools.register(builder.build(), subsystem,
ModelVersion.create(1, 0, 0));
    }
}
```

Now say we want to do a new version of the model. This new version contains a new attribute called 'new-attr' which cannot be defined when transforming to 1.1.0, we bump the model version to 2.0.0:

```
public class SomeExtension implements Extension {

    private static final String SUBSYSTEM_NAME = "my-subsystem"'

    private static final int MANAGEMENT_API_MAJOR_VERSION = 2;
    private static final int MANAGEMENT_API_MINOR_VERSION = 0;
    private static final int MANAGEMENT_API_MICRO_VERSION = 0;

    @Override
    public void initialize(ExtensionContext context) {
        SubsystemRegistration registration = context.registerSubsystem(SUBSYSTEM_NAME,
MANAGEMENT_API_MAJOR_VERSION,
                MANAGEMENT_API_MINOR_VERSION, MANAGEMENT_API_MICRO_VERSION);
        //Register the resource definitions
        ....
    }
```

There are a few ways to evolve your transformers:

- The old way
- Chained transformers

The old way

This is the way that has been used up to WildFly 8.x. However, in WildFly 9 and later, it is strongly recommended to migrate to what is mentioned in Chained transformers

Now we need some new transformers from the current ModelVersion to 1.1.0 where we reject any defined occurrances of our new attribute `new-attr`:

```
private void registerTransformers(final SubsystemRegistration subsystem) {
        registerTransformers_1_0_0(subsystem);
        registerTransformers_1_1_0(subsystem);
    }

    /**
     * Registers transformers from the current version to ModelVersion 1.1.0
     */
    private void registerTransformers_1_1_0(SubsystemRegistration subsystem) {
        ResourceTransformationDescriptionBuilder builder =
TransformationDescriptionBuilder.Factory.createSubsystemInstance();
        builder.getAttributeBuilder()
            .addRejectCheck(RejectAttributeChecker.DEFINED, "new-attr")
            .end();
        TransformationDescription.Tools.register(builder.build(), subsystem,
ModelVersion.create(1, 1, 0));
    }
```

So that is all well and good, however we also need to take into account that `new-attr` **does not exist in ModelVersion 1.0.0 either**, so we need to extend our transformer for 1.0.0 to reject it there as well. As you can see 1.0.0 also rejects a defined 'attr1' in addition to the 'new-attr'(which is rejected in both versions).

```
/**
     * Registers transformers from the current version to ModelVersion 1.0.0
     */
    private void registerTransformers_1_0_0(SubsystemRegistration subsystem) {
        ResourceTransformationDescriptionBuilder builder =
TransformationDescriptionBuilder.Factory.createSubsystemInstance();
        builder.getAttributeBuilder()
            .addRejectCheck(RejectAttributeChecker.DEFINED, "attr1", "new-attr")
            .end();
        TransformationDescription.Tools.register(builder.build(), subsystem,
ModelVersion.create(1, 0, 0));
    }
}
```

Now `new-attr` will be rejected if defined for all previous model versions.

Chained transformers

Since 'The old way' had a lot of duplication of code, since WildFly 9 we now have chained transformers. You obtain a `ChainedTransformationDescriptionBuilder` which is a different entry point to the `ResourceTransformationDescriptionBuilder` we have seen earlier. Each `ResourceTransformationDescriptionBuilder` deals with transformation across one version delta.

```
private void registerTransformers(SubsystemRegistration subsystem) {
        ModelVersion version1_1_0 = ModelVersion.create(1, 1, 0);
        ModelVersion version1_0_0 = ModelVersion.create(1, 0, 0);

        ChainedTransformationDescriptionBuilder chainedBuilder =

TransformationDescriptionBuilder.Factory.createChainedSubystemInstance(subsystem.getSubsystemVersi
//Differences between the current version and 1.1.0
        ResourceTransformationDescriptionBuilder builder110 =
            chainedBuilder.create(subsystem.getSubsystemVersion(), version1_1_0);
        builder110.getAttributeBuilder()
            .addRejectCheck(RejectAttributeChecker.DEFINED, "new-attr")
            .end();

        //Differences between the 1.1.0 and 1.0.0
        ResourceTransformationDescriptionBuilder builder100 =
            chainedBuilder.create(subsystem.getSubsystemVersion(), version1_0_0);
        builder110.getAttributeBuilder()
            .addRejectCheck(RejectAttributeChecker.DEFINED, "attr1")
            .end();

        chainedBuilder.buildAndRegister(subsystem, new ModelVersion[]{version1_0_0,
version1_1_0});
```

The `buildAndRegister(ModelVersion[]... chains)` method registers a chain consisting of the built `builder110` and `builder100` for transformation to 1.0.0, and a chain consisting of the built `builder110` for transformation to 1.1.0. It allows you to specify more than one chain.

Now when transforming from the current version to 1.0.0, the resource is first transformed from the current version to 1.1.0 (which rejects a defined `new-attr`) and then it is transformed from 1.1.0 to 1.0.0 (which rejects a defined `attr1`). So when evolving transformers you should normally only need to add things to the last version delta. The full current-to-1.1.0 transformation is run before the 1.1.0-to-1.0.0 transformation is run.

One thing worth pointing out that the value returned by `TransformationContext.readResource(PathAddress address)` and `TransformationContext.readResourceFromRoot(PathAddress address)` which you can use from your custom `RejectAttributeChecker`, `DiscardAttributeChecker` and `AttributeConverter` behaves slightly differently depending on if you are transforming an operation or a resource.

During *resource transformation* this will be the latest model, so in our above example, in the current-to-1.1.0 transformation it will be the original model. In the 1.1.0-to-1.0.0 transformation, it will be the result of the current-to-1.1.0 transformation.

During *operation transformation* these methods will always return the original model (we are transforming operations, not resources!).

In WildFly 9 we are now less aggressive about transforming to all previous versions of WildFly, however we still have a lot of good tests for running against 7.1.x, 8. Also, for Red Hat employees we have tests against EAP versions. These tests no longer get run by default, to run them you need to specify some system properties when invoking maven. They are:

- `-Djboss.test.transformers.subsystem.old` - enables the non-default subsystem tests.
- -Djboss.test.transformers.eap - (Red Hat developers only), enables the eap tests, but only the ones run by default. If run in conjunction with `-Djboss.test.transformers.subsystem.old` you get all the possible subsystem tests run.
- -Djboss.test.transformers.core.old - enables the non-default core model tests.

11.5.8 Testing transformers

To test transformation you need to extend `org.jboss.as.subsystem.test.AbstractSubsystemTest` or `org.jboss.as.subsystem.test.AbstractSubsystemBaseTest`. Then, in order to have the best test coverage possible, you should test the fullest configuration that will work, and you should also test configurations that don't work if you have rejecting transformers registered. The following example is from the threads subsystem, and I have only included the tests against 7.1.2 - there are more! First we need to set up our test:

```
public class ThreadsSubsystemTestCase extends AbstractSubsystemBaseTest {
    public ThreadsSubsystemTestCase() {
        super(ThreadsExtension.SUBSYSTEM_NAME, new ThreadsExtension());
    }

    @Override
    protected String getSubsystemXml() throws IOException {
        return readResource("threads-subsystem-1_1.xml");
    }
```

So we say that this test is for the `threads` subsystem, and that it is implemented by `ThreadsExtension`. This is the same test framework as we use in Example subsystem#Testing the parsers, but we will only talk about the parts relevant to transformers here.

Testing a configuration that works

To test a configuration xxx

```
@Test
    public void testTransformerAS712() throws Exception {
        testTransformer_1_0(ModelTestControllerVersion.V7_1_2_FINAL);
    }
    /**
     * Tests transformation of model from 1.1.0 version into 1.0.0 version.
     *
     * @throws Exception
     */
    private void testTransformer_1_0(ModelTestControllerVersion controllerVersion) throws
Exception {
        String subsystemXml = "threads-transform-1_0.xml";   //This has no expressions not
understood by 1.0
        ModelVersion modelVersion = ModelVersion.create(1, 0, 0); //The old model version
        //Use the non-runtime version of the extension which will happen on the HC
        KernelServicesBuilder builder =
createKernelServicesBuilder(AdditionalInitialization.MANAGEMENT)
                .setSubsystemXmlResource(subsystemXml);

        final PathAddress subsystemAddress =
PathAddress.pathAddress(PathElement.pathElement(SUBSYSTEM, mainSubsystemName));

        // Add legacy subsystems
        builder.createLegacyKernelServicesBuilder(null, controllerVersion, modelVersion)
                .addOperationValidationResolve("add",
subsystemAddress.append(PathElement.pathElement("thread-factory")))
                .addMavenResourceURL("org.jboss.as:jboss-as-threads:" +
controllerVersion.getMavenGavVersion())
                .excludeFromParent(SingleClassFilter.createFilter(ThreadsLogger.class));

        KernelServices mainServices = builder.build();
        KernelServices legacyServices = mainServices.getLegacyServices(modelVersion);
        Assert.assertNotNull(legacyServices);
        checkSubsystemModelTransformation(mainServices, modelVersion);
    }
```

What this test does is get the builder to configure the test controller using `threads-transform-1_0.xml`. This main builder works with the current subsystem version, and the jars in the WildFly checkout.

Next we configure a 'legacy' controller. This will run the version of the core libraries (e.g the `controller` module) as found in the targeted legacy version of JBoss AS/Wildfly), and the subsystem. We need to pass in that it is using the core AS version 7.1.2.Final (i.e. the `ModelTestControllerVersion.V7_1_2_FINAL` part) and that that version is ModelVersion 1.0.0. Next we have some `addMavenResourceURL()` calls passing in the Maven GAVs of the old version of the subsystem and any dependencies it has needed to boot up. Normally, specifying just the Maven GAV of the old version of the subsystem is enough, but that depends on your subsystem. In this case the old subsystem GAV is enough. When booting up the legacy controller the framework uses the parsed operations from the main controller and transforms them using the 1.0.0 transformer in the threads subsystem. The `addOperationValidationResolve()` and `excludeFromParent()` calls are not normally necessary, see the javadoc for more examples.

The call to `KernelServicesBuilder.build()` will build both the main controller and the legacy controller. As part of that it also boots up a second copy of the main controller using the transformed operations to make sure that the 'old' ops to boot our subsystem will still work on the current controller, which is important for backwards compatibility of CLI scripts. To tweak how that is done if you see failures there, see `LegacyKernelServicesInitializer.skipReverseControllerCheck()` and `LegacyKernelServicesInitializer.configureReverseControllerCheck()`. The `LegacyKernelServicesInitializer` is what gets returned by `KernelServicesBuilder.createLegacyKernelServicesBuilder()`.

Finally we call `checkSubsystemModelTransformation()` which reads the full legacy subsystem model. The legacy subsystem model will have been built up from the transformed boot operations from the parsed xml. The operations get transformed by the operation transformers. Then it takes the model of the current subsystem and transforms that using the resource transformers. Then it compares the two models, which should be the same. In some rare cases it is not possible to get those two models exactly the same, so there is a version of this method that takes a `ModelFixer` to make adjustments. The `checkSubsystemModelTransformation()` method also makes sure that the legacy model is valid according to the legacy subsystem's resource definition.

The legacy subsystem resource definitions are read on demand from the legacy controller when the tests run. In some older versions of subsystems (before we converted everything to use ResourceDefinition, and DescriptionProvider implementations were coded by hand) there were occasional problems with the resource definitions and they needed to be touched up. In this case you can generate a new one, touch it up and store the result in a file in the test resources under `/same/package/as/the/test/class/{{subsystem-name-model-version`. This will then prefer the file read from the file system to the one read at runtime. To generate the .dmr file, you need to generate it by adding a temporary test (make sure that you adjust `controllerVersion` and `modelVersion` to what you want to generate):

```
@Test
    public void deleteMeWhenDone() throws Exception {
        ModelTestControllerVersion controllerVersion = ModelTestControllerVersion.V7_1_2_FINAL;
        ModelVersion modelVersion = ModelVersion.create(1, 0, 0);
        KernelServicesBuilder builder = createKernelServicesBuilder(null);

        builder.createLegacyKernelServicesBuilder(null, controllerVersion, modelVersion)
            .addMavenResourceURL("org.jboss.as:jboss-as-threads:" +
controllerVersion.getMavenGavVersion());
        KernelServices services = builder.build();

        generateLegacySubsystemResourceRegistrationDmr(services, modelVersion);
    }
```

Now run the test and delete it. The legacy .dmr file should be in `target/test-classes/org/jboss/as/subsystem/test/<your-subsystem-name>-<your-vers`. Copy this .dmr file to the correct location in your project's test resources.

Testing a configuration that does not work

The `threads` subsystem (like several others) did not support the use of expression values in the version that came with JBoss AS 7.1.2.Final. So we have a test that attempts to use expressions, and then fixes each resource and attribute where expressions were not allowed.

```
@Test
    public void testRejectExpressionsAS712() throws Exception {
        testRejectExpressions_1_0_0(ModelTestControllerVersion.V7_1_2_FINAL);
    }

    private void testRejectExpressions_1_0_0(ModelTestControllerVersion controllerVersion)
throws Exception {
        // create builder for current subsystem version
        KernelServicesBuilder builder =
createKernelServicesBuilder(createAdditionalInitialization());

        // create builder for legacy subsystem version
        ModelVersion version_1_0_0 = ModelVersion.create(1, 0, 0);
        builder.createLegacyKernelServicesBuilder(null, controllerVersion, version_1_0_0)
                .addMavenResourceURL("org.jboss.as:jboss-as-threads:" +
controllerVersion.getMavenGavVersion())
                .excludeFromParent(SingleClassFilter.createFilter(ThreadsLogger.class));

        KernelServices mainServices = builder.build();
        KernelServices legacyServices = mainServices.getLegacyServices(version_1_0_0);

        Assert.assertNotNull(legacyServices);
        Assert.assertTrue("main services did not boot", mainServices.isSuccessfulBoot());
        Assert.assertTrue(legacyServices.isSuccessfulBoot());

        List<ModelNode> xmlOps = builder.parseXmlResource("expressions.xml");

        ModelTestUtils.checkFailedTransformedBootOperations(mainServices, version_1_0_0, xmlOps,
getConfig());
    }
```

Again we boot up a current and a legacy controller. However, note in this case that they are both empty, no xml was parsed on boot so there are no operations to boot up the model. Instead once the controllers have been booted, we call `KernelServicesBuilder.parseXmlResource()` which gets the operations from `expressions.xml`. `expressions.xml` uses expressions in all the places they were not allowed in 7.1.2.Final. For each resource `ModelTestUtils.checkFailedTransformedBootOperations()` will check that the `add` operation gets rejected, and then correct one attribute at a time until the resource has been totally corrected. Once the `add` operation is totally correct, it will check that the add operation no longer is rejected. The configuration for this is the `FailedOperationTransformationConfig` returned by the `getConfig()` method:

```
private FailedOperationTransformationConfig getConfig() {
        PathAddress subsystemAddress = PathAddress.pathAddress(ThreadsExtension.SUBSYSTEM_PATH);
        FailedOperationTransformationConfig.RejectExpressionsConfig allowedAndKeepalive =
                new
FailedOperationTransformationConfig.RejectExpressionsConfig(PoolAttributeDefinitions.ALLOW_CORE_TI
PoolAttributeDefinitions.KEEPALIVE_TIME);
...
        return new FailedOperationTransformationConfig()

.addFailedAttribute(subsystemAddress.append(PathElement.pathElement(CommonAttributes.BLOCKING_BOUN
allowedAndKeepalive)

.addFailedAttribute(subsystemAddress.append(PathElement.pathElement(CommonAttributes.BOUNDED_QUEUE
allowedAndKeepalive)
    }
```

So what this means is that we expect the `allow-core-timeout` and `keepalive-time` attributes for the `blocking-bounded-queue-thread-pool=*` and `bounded-queue-thread-pool=*` add operations to use expressions in the parsed xml. We then expect them to fail since there should be transformers in place to reject expressions, and correct them one at a time until the add operation should pass. As well as doing the `add` operations the `ModelTestUtils.checkFailedTransformedBootOperations()` method will also try calling `write-attribute` for each attribute, correcting as it goes along. As well as allowing you to test rejection of expressions `FailedOperationTransformationConfig` also has some helper classes to help testing rejection of other scenarios.

11.5.9 Common transformation use-cases

Most transformations are quite similar, so this section covers some of the actual transformation patterns found in the WildFly codebase. We will look at the output of CompareModelVersionsUtil, and see what can be done to transform for the older slave HCs. The examples come from the WildFly codebase but are stripped down to focus solely on the use-case being explained in an attempt to keep things as clear/simple as possible.

Child resource type does not exist in legacy model

Looking at the model comparison between WildFly and JBoss AS 7.2.0, there is a change to the `remoting` subsystem. The relevant part of the output is:

```
====== Resource root address: ["subsystem" => "remoting"] - Current version: 2.0.0; legacy
version: 1.2.0 =======
--- Problems for relative address to root []:
Missing child types in current: []; missing in legacy [http-connector]
```

So our current model has added a child type called `http-connector` which was not there in 7.2.0. This is configurable, and adds new behavior, so it can not be part of a configuration sent across to a legacy slave running version 1.2.0. So we add the following to `RemotingExtension` to reject all instances of that child type against ModelVersion 1.2.0.

```
@Override
    public void initialize(ExtensionContext context) {
        ....
        if (context.isRegisterTransformers()) {
            registerTransformers_1_1(registration);
            registerTransformers_1_2(registration);
        }
    }

    private void registerTransformers_1_2(SubsystemRegistration registration) {
        TransformationDescription.Tools.register(get1_2_0_1_3_0Description(), registration,
VERSION_1_2);
    }

    private static TransformationDescription get1_2_0_1_3_0Description() {
        ResourceTransformationDescriptionBuilder builder =
ResourceTransformationDescriptionBuilder.Factory.createSubsystemInstance();
        builder.rejectChildResource(HttpConnectorResource.PATH);

        return builder.build();
    }
```

Since this child resource type also does not exist in ModelVersion 1.1.0 we need to reject it there as well using a similar mechanism.

Attribute does not exist in the legacy subsystem

Default value of the attribute is the same as legacy implied behavior

This example also comes from the `remoting` subsystem, and is probably the most common type of transformation. The comparison tells us that there is now an attribute under `/subsystem=remoting/remote-outbound-connection=*` called `protocol` which did not exist in the older version:

```
====== Resource root address: ["subsystem" => "remoting"] - Current version: 2.0.0; legacy
version: 1.2.0 =======
--- Problems for relative address to root []:
....
--- Problems for relative address to root ["remote-outbound-connection" => "*"]:
Missing attributes in current: []; missing in legacy [protocol]
Missing parameters for operation 'add' in current: []; missing in legacy [protocol]
```

This difference also affects the `add` operation. Looking at the current model the valid values for the `protocol` attribute are `remote`, `http-remoting` and `https-remoting`. The last two are new protocols introduced in WildFly 8, meaning that the *implied behaviour* in JBoss 7.2.0 and earlier is the `remote` protocol. Since this attribute does not exist in the legacy model we want to discard this attribute if it is `undefined` or if it has the value `remote`, both of which are in line with what the legacy slave HC is hardwired to use. Also we want to reject it if it has a value different from `remote`. So what we need to do when registering transformers against ModelVersion 1.2.0 to handle this attribute:

```
private void registerTransformers_1_2(SubsystemRegistration registration) {
        TransformationDescription.Tools.register(get1_2_0_1_3_0Description(), registration,
VERSION_1_2);
    }

    private static TransformationDescription get1_2_0_1_3_0Description() {
        ResourceTransformationDescriptionBuilder builder =
ResourceTransformationDescriptionBuilder.Factory.createSubsystemInstance();

protocolTransform(builder.addChildResource(RemoteOutboundConnectionResourceDefinition.ADDRESS)
                .getAttributeBuilder());
        return builder.build();
    }

    private static AttributeTransformationDescriptionBuilder
protocolTransform(AttributeTransformationDescriptionBuilder builder) {
        builder.setDiscard(new DiscardAttributeChecker.DiscardAttributeValueChecker(new
ModelNode(Protocol.REMOTE.toString())), RemoteOutboundConnectionResourceDefinition.PROTOCOL)
                .addRejectCheck(RejectAttributeChecker.DEFINED,
RemoteOutboundConnectionResourceDefinition.PROTOCOL);
        return builder;
    }
```

So the first thing to happens is that we register a `DiscardAttributeChecker.DiscardAttributeValueChecker` which discards the attribute if it is either `undefined` (the default value in the current model is `remote`), or `defined` and has the value `remote`. Remembering that the `discard` phase always happens before the `reject` phase, the reject checker checks that the `protocol` attribute is defined, and rejects it if it is. The only reason it would be `defined` in the reject check, is if it was not discarded by the discard check. Hopefully this example shows that the discard and reject checkers often work in pairs.

An alternative way to write the `protocolTransform()` method would be:

```
private static AttributeTransformationDescriptionBuilder
protocolTransform(AttributeTransformationDescriptionBuilder builder) {
        builder.setDiscard(new DiscardAttributeChecker.DefaultDiscardAttributeChecker() {
                    @Override
                    protected boolean isValueDiscardable(final PathAddress address, final String
attributeName, final ModelNode attributeValue, final TransformationCon
                        return !attributeValue.isDefined() ||
attributeValue.asString().equals(Protocol.REMOTE.toString());
                    }
                }, RemoteOutboundConnectionResourceDefinition.PROTOCOL)
         .addRejectCheck(RejectAttributeChecker.DEFINED,
RemoteOutboundConnectionResourceDefinition.PROTOCOL);
         return builder;
```

The reject check remains the same, but we have implemented the discard check by using `DiscardAttributeChecker.DefaultDiscardAttributeChecker` instead. However, the effect of the discard check is exactly the same as when we used `DiscardAttributeChecker.DiscardAttributeValueChecker`.

Default value of the attribute is different from legacy implied behaviour

We touched on this in the weld subsystem example we used earlier in this guide, but let's take a more thorough look. Our comparison tells us that we have two new attributes `require-bean-descriptor` and `non-portable-mode`:

```
====== Resource root address: ["subsystem" => "weld"] - Current version: 2.0.0; legacy version:
1.0.0 =======
--- Problems for relative address to root []:
Missing attributes in current: []; missing in legacy [require-bean-descriptor,
non-portable-mode]
Missing parameters for operation 'add' in current: []; missing in legacy
[require-bean-descriptor, non-portable-mode]
```

Now when we look at this we see that the default value for both of the attributes in the current model is `false`, which allows us more flexible behavior introduced in CDI 1.1 (which was introduced with this version of the subsystem). The old model does not have these attributes, and implements CDI 1.0, which under the hood (using our weld subsystem expertise knowledge) implies the values `true` for both of these. So our transformer must reject anything that is not `true` for these attributes. Let us look at the transformer registered by the WeldExtension:

```
private void registerTransformers(SubsystemRegistration subsystem) {
        ResourceTransformationDescriptionBuilder builder =
TransformationDescriptionBuilder.Factory.createSubsystemInstance();
        //These new attributes are assumed to be 'true' in the old version but default to false
in the current version. So discard if 'true' and reject if 'undefined'.
        builder.getAttributeBuilder()
                .setDiscard(new DiscardAttributeChecker.DiscardAttributeValueChecker(false,
false, new ModelNode(true)),
                        WeldResourceDefinition.NON_PORTABLE_MODE_ATTRIBUTE,
WeldResourceDefinition.REQUIRE_BEAN_DESCRIPTOR_ATTRIBUTE)
                .addRejectCheck(new RejectAttributeChecker.DefaultRejectAttributeChecker() {

                    @Override
                    public String getRejectionLogMessage(Map<String, ModelNode> attributes) {
                        return
WeldMessages.MESSAGES.rejectAttributesMustBeTrue(attributes.keySet());
                    }

                    @Override
                    protected boolean rejectAttribute(PathAddress address, String attributeName,
ModelNode attributeValue,
                            TransformationContext context) {
                        //This will not get called if it was discarded, so reject if it is
undefined (default==false) or if defined and != 'true'
                        return !attributeValue.isDefined() ||
!attributeValue.asString().equals("true");
                    }
                }, WeldResourceDefinition.NON_PORTABLE_MODE_ATTRIBUTE,
WeldResourceDefinition.REQUIRE_BEAN_DESCRIPTOR_ATTRIBUTE)
                .end();
        TransformationDescription.Tools.register(builder.build(), subsystem,
ModelVersion.create(1, 0, 0));
    }
```

This looks a bit more scary than the previous transformer we have seen, but isn't actually too bad. The first thing we do is register a `DiscardAttributeChecker.DiscardAttributeValueChecker` which will discard the attribute if it has the value `true`. It will not discard if it is `undefined` since that defaults to `false`. This is registered for both attributes.

If the attributes had the value `true` they will get discarded we will not hit the reject checker since discarded attributes never get checked for rejection. If on the other hand they were an expression (since we are interested in the actual value, but cannot evaluate what value an expression will resolve to on the target from the DC running the transformers), `false`, or `undefined` (which will then default to `false`) they will not get discarded and will need to be rejected. So our `RejectAttributeChecker.DefaultRejectAttributeChecker.rejectAttribute()` method will return `true` (i.e. reject) if the attribute value is `undefined` (since that defaults to `false`) or if it is defined and 'not equal to `true`'. It is better to check for 'not equal to `true`' than to check for 'equal to `false`' since if an expression was used we still want to reject, and only the 'not equal to `true`' check would actually kick in in that case.

The other thing we need in our `DiscardAttributeChecker.DiscardAttributeValueChecker` is to override the `getRejectionLogMessage()` method to get the message to be displayed when rejecting the transformation. In this case it says something along the lines "These attributes must be 'true' for use with CDI 1.0 '%s'", with the names of the attributes having been rejected substituting the `%s`.

Attribute has a different default value

– TODO

(The gist of this is to use a value converter, such that if the attribute is undefined, and hence the default value will take effect, then the value gets converted to the current version's default value. This ensures that the legacy HC will use the same effective setting as current version HCs.

Note however that a change in default values is a form of incompatible API change, since CLI scripts written assuming the old defaults will now produce a configuration that behaves differently. Transformers make it possible to have a consistently configured domain even in the presence of this kind of incompatible change, but that doesn't mean such changes are good practice. They are generally unacceptable in WildFly's own subsystems.

One trick to ameliorate the impact of a default value change is to modify the xml parser for the **old** schema version such that if the xml attribute is not configured, the parser sets the old default value for the attribute, instead of `undefined`. This approach allows the parsing of old config documents to produce results consistent with what happened when they were created. It does not help with CLI scripts though.)

Attribute has a different type

Here the example comes from the `capacity` parameter some way into the `modcluster` subsystem, and the legacy version is AS 7.1.2.Final. There are quite a few differences, so I am only showing the ones relevant for this example:

```
====== Resource root address: ["subsystem" => "modcluster"] - Current version: 2.0.0; legacy
version: 1.2.0 =======
...
--- Problems for relative address to root ["mod-cluster-config" =>
"configuration","dynamic-load-provider" => "configuration","custom-load-m
etric" => "*"]:
Different 'type' for attribute 'capacity'. Current: DOUBLE; legacy: INT
Different 'expressions-allowed' for attribute 'capacity'. Current: true; legacy: false
...
Different 'type' for parameter 'capacity' of operation 'add'. Current: DOUBLE; legacy: INT
Different 'expressions-allowed' for parameter 'capacity' of operation 'add'. Current: true;
legacy: false
```

So as we can see expressions are not allowed for the `capacity` attribute, and the current type is `double` while the legacy subsystem is `int`. So this means that if the value is for example `2.0` we can convert this to `2`, but `2.5` cannot be converted. The way this is solved in the ModClusterExtension is to register the following some other attributes are registered here, but hopefully it is clear anyway:

```
dynamicLoadProvider.addChildResource(LOAD_METRIC_PATH)
                    .getAttributeBuilder()
                        .addRejectCheck(RejectAttributeChecker.SIMPLE_EXPRESSIONS, TYPE, WEIGHT,
CAPACITY, PROPERTY)
                        .addRejectCheck(CapacityCheckerAndConverter.INSTANCE, CAPACITY)
                        .setValueConverter(CapacityCheckerAndConverter.INSTANCE, CAPACITY)
                        ...
                        .end();
```

So we register that we should reject expressions, and we also register the `CapacityCheckerAndConverter` for `capacity`. `CapacityCheckerAndConverter` extends the convenience class `DefaultCheckersAndConverter` which implements the `DiscardAttributeChecker`, `RejectAttributeChecker`, and `AttributeConverter` interfaces. We have seen `DiscardAttributeChecker` and `RejectAttributeChecker` in previous examples. Since we now need to convert a value we need an instance of `AttributeConverter`.

```
static class CapacityCheckerAndConverter extends DefaultCheckersAndConverter {

        static final CapacityCheckerAndConverter INSTANCE = new CapacityCheckerAndConverter();
```

We should not discard so `isValueDiscardable()` from `DiscardAttributeChecker` always returns `false`:

```
@Override
        protected boolean isValueDiscardable(PathAddress address, String attributeName,
ModelNode attributeValue, TransformationContext context) {
            //Not used for discard
            return false;
        }

        @Override
        public String getRejectionLogMessage(Map<String, ModelNode> attributes) {
            return
ModClusterMessages.MESSAGES.capacityIsExpressionOrGreaterThanIntegerMaxValue(attributes.get(CAPACI
}
```

Now we check to see if we can convert the attribute to an `int` and reject if not. Note that if it is an expression, we have no idea what its value will resolve to on the target host, so we need to reject it. Then we try to change it into an `int`, and reject if that was not possible:

```
@Override
        protected boolean rejectAttribute(PathAddress address, String attributeName, ModelNode
attributeValue, TransformationContext context) {
            if (checkForExpression(attributeValue)
                    || (attributeValue.isDefined() &&
!isIntegerValue(attributeValue.asDouble()))) {
                return true;
            }
            Long converted = convert(attributeValue);
            return (converted != null && (converted > Integer.MAX_VALUE || converted <
Integer.MIN_VALUE));
        }
```

And then finally we do the conversion:

```
@Override
        protected void convertAttribute(PathAddress address, String attributeName, ModelNode
attributeValue, TransformationContext context) {
            Long converted = convert(attributeValue);
            if (converted != null && converted <= Integer.MAX_VALUE && converted >=
Integer.MIN_VALUE) {
                attributeValue.set((int)converted.longValue());
            }
        }


        private Long convert(ModelNode attributeValue) {
            if (attributeValue.isDefined() && !checkForExpression(attributeValue)) {
                double raw = attributeValue.asDouble();
                if (isIntegerValue(raw)) {
                    return Math.round(raw);
                }
            }
            return null;
        }

        private boolean isIntegerValue(double raw) {
            return raw == Double.valueOf(Math.round(raw)).doubleValue();
        }

    }
```

11.6 Key Interfaces and Classes Relevant to Extension Developers

In the first major section of this guide, we provided an example of how to implement an extension to the AS. The emphasis there was learning by doing. In this section, we'll focus a bit more on the major WildFly interfaces and classes that most are relevant to extension developers. The best way to learn about these interfaces and classes in detail is to look at their javadoc. What we'll try to do here is provide a brief introduction of the key items and how they relate to each other.

Before digging into this section, readers are encouraged to read the "Core Management Concepts" section of the Admin Guide.

11.6.1 Extension Interface

The `org.jboss.as.controller.Extension` interface is the hook by which your extension to the AS kernel is able to integrate with the AS. During boot of the AS, when the `<extension>` element in the AS's xml configuration file naming your extension is parsed, the JBoss Modules module named in the element's name attribute is loaded. The standard JDK `java.lang.ServiceLoader` mechanism is then used to load your module's implementation of this interface.

The function of an `Extension` implementation is to register with the core AS the management API, xml parsers and xml marshallers associated with the extension module's subsystems. An `Extension` can register multiple subsystems, although the usual practice is to register just one per extension.

Once the `Extension` is loaded, the core AS will make two invocations upon it:

- `void initializeParsers(ExtensionParsingContext context)`

When this is invoked, it is the `Extension` implementation's responsibility to initialize the XML parsers for this extension's subsystems and register them with the given `ExtensionParsingContext`. The parser's job when it is later called is to create `org.jboss.dmr.ModelNode` objects representing WildFly management API operations needed make the AS's running configuration match what is described in the xml. Those management operation `ModelNode` s are added to a list passed in to the parser.

A parser for each version of the xml schema used by a subsystem should be registered. A well behaved subsystem should be able to parse any version of its schema that it has ever published in a final release.

- `void initialize(ExtensionContext context)`

When this is invoked, it is the `Extension` implementation's responsibility to register with the core AS the management API for its subsystems, and to register the object that is capable of marshalling the subsystem's in-memory configuration back to XML. Only one XML marshaller is registered per subsystem, even though multiple XML parsers can be registered. The subsystem should always write documents that conform to the latest version of its XML schema.

The registration of a subsystem's management API is done via the `ManagementResourceRegistration` interface. Before discussing that interface in detail, let's describe how it (and the related `Resource` interface) relate to the notion of managed resources in the AS.

11.6.2 WildFly Managed Resources

Each subsystem is responsible for managing one or more management resources. The conceptual characteristics of a management resource are covered in some detail in the Admin Guide; here we'll just summarize the main points. A management resource has

- An ***address*** consisting of a list of key/value pairs that uniquely identifies a resource
- Zero or more ***attributes***, the value of which is some sort of `org.jboss.dmr.ModelNode`
- Zero or more supported ***operations***. An operation has a string name and zero or more parameters, each of which is a key/value pair where the key is a string naming the parameter and the value is some sort of `ModelNode`
- Zero or more ***children***, each of which in turn is a managed resource

The implementation of a managed resource is somewhat analogous to the implementation of a Java object. A managed resource will have a "type", which encapsulates API information about that resource and logic used to implement that API. And then there are actual instances of the resource, which primarily store data representing the current state of a particular resource. This is somewhat analogous to the "class" and "object" notions in Java.

A managed resource's type is encapsulated by the `org.jboss.as.controller.registry.ManagementResourceRegistration` the core AS creates when the type is registered. The data for a particular instance is encapsulated in an implementation of the `org.jboss.as.controller.registry.Resource` interface.

11.6.3 ManagementResourceRegistration Interface

In the Java analogy used above, the `ManagementResourceRegistration` is analogous to the "class", while the `Resource` discussed below is analogous to an instance of that class.

A `ManagementResourceRegistration` represents the specification for a particular managed resource type. All resources whose address matches the same pattern will be of the same type, specified by the type's `ManagementResourceRegistration`. The MRR encapsulates:

- A `PathAddress` showing the address pattern that matches resources of that type. This `PathAddress` can and typically does involve wildcards in the value of one or more elements of the address. In this case there can be more than one instance of the type, i.e. different `Resource` instances.
- Definition of the various attributes exposed by resources of this type, including the `OperationStepHandler` implementations used for reading and writing the attribute values.
- Definition of the various operations exposed by resources of this type, including the `OperationStepHandler` implementations used for handling user invocations of those operations.
- Definition of child resource types. `ManagementResourceRegistration` instances form a tree.
- Definition of management notifications emitted by resources of this type.
- Definition of capabilities provided by resources of this type.
- Definition of RBAC access constraints that should be applied by the management kernel when authorizing operations against resources of this type.
- Whether the resource type is an alias to another resource type, and if so information about that relationship. Aliases are primarily used to preserve backwards compatibility of the management API when the location of a given type of resources is moved in a newer release.

The `ManagementResourceRegistration` interface is a subinterface of `ImmutableManagementResourceRegistration`, which provides a read-only view of the information encapsulated by the MRR. The MRR subinterface adds the methods needed for registering the attributes, operations, children, etc.

Extension developers do not directly instantiate an MRR. Instead they create a `ResourceDefinition` for the root resource type for each subsystem, and register it with the `ExtensionContext` passed in to their `Extension` implementation's `initialize` method:

```
public void initialize(ExtensionContext context) {
        SubsystemRegistration subsystem = context.registerSubsystem(SUBSYSTEM_NAME,
CURRENT_VERSION);
        subsystem.registerXMLElementWriter(getOurXmlWriter());
        ResourceDefinition rd = getOurSubsystemDefinition();
        ManagementResourceRegistration mrr = subsystem.registerSubsystemModel(rd));
    }
```

The kernel uses the provided `ResourceDefinition` to construct a `ManagementResourceRegistration` and then passes that MRR to the various `registerXXX` methods implemented by the `ResourceDefinition`, giving it the change to record the resource type's attributes, operations and children.

11.6.4 ResourceDefinition Interface

An implementation of `ResourceDefinition` is the primary class used by an extension developer when defining a managed resource type. It provides basic information about the type, exposes a `DescriptionProvider` used to generate a DMR description of the type, and implements callbacks the kernel can invoke when building up the `ManagementResourceRegistration` to ask for registration of definitions of attributes, operations, children, notifications and capabilities.

Almost always an extension author will create their `ResourceDefinition` by creating a subclass of the `org.jboss.as.controller.SimpleResourceDefinition` class or of its `PersistentResourceDefinition` subclass. Both of these classes have constructors that take a `Parameters` object, which is a simple builder class to use to provide most of the key information about the resource type. The extension-specific subclass would then take responsibility for any additional behavior needed by overriding the `registerAttributes`, `registerOperations`, `registerNotifications` and `registerChildren` callbacks to do whatever is needed beyond what is provided by the superclasses.

For example, to add a writable attribute:

```
@Override
    public void registerAttributes(ManagementResourceRegistration resourceRegistration) {
        super.registerAttributes(resourceRegistration);
        // Now we register the 'foo' attribute
        AttributeDefinition ad = FOO; // constant declared elsewhere
        OperationStepHandler writeHandler = new FooWriteAttributeHandler();
        resourceRegistration.registerReadWriteHandler(ad, null, writeHandler); // null read
handler means use default read handling
    }
```

To register a custom operation:

```
@Override
    public void registerOperations(ManagementResourceRegistration resourceRegistration) {
        super.registerOperations(resourceRegistration);
        // Now we register the 'foo-bar' custom operation
        OperationDefinition od = FooBarOperationStepHandler.getDefinition();
        OperationStepHandler osh = new FooBarOperationStepHandler();
        resourceRegistration.registerOperationHandler(od, osh);
    }
```

To register a child resource type:

```
@Override
    public void registerChildren(ManagementResourceRegistration resourceRegistration) {
        super.registerChildren(resourceRegistration);
        // Now we register the 'baz=*' child type
        ResourceDefinition rd = new BazResourceDefinition();
        resourceRegistration.registerSubmodel(rd);
    }
```

ResourceDescriptionResolver

One of the things a `ResourceDefinition` must be able to do is provide a `DescriptionProvider` that provides a proper DMR description of the resource to use as the output for the standard `read-resource-description` management operation. Since you are almost certainly going to be using one of the standard `ResourceDefinition` implementations like `SimpleResourceDefinition`, the creation of this `DescriptionProvider` is largely handled for you. The one thing that is not handled for you is providing the localized free form text descriptions of the various attributes, operations, operation parameters, child types, etc used in creating the resource description.

For this you must provide an implementation of the `ResourceDescriptionResolver` interface, typically passed to the `Parameters` object provided to the `SimpleResourceDefinition` constructor. This interface has various methods that are invoked when a piece of localized text description is needed.

Almost certainly you'll satisfy this requirement by providing an instance of the `StandardResourceDescriptionResolver` class.

`StandardResourceDescriptionResolver` uses a `ResourceBundle` to load text from a properties file available on the classpath. The keys in the properties file must follow patterns expected by `StandardResourceDescriptionResolver`. See the `StandardResourceDescriptionResolver` javadoc for further details.

The biggest task here is to create the properties file and add the text descriptions. A text description must be provided for everything. The typical thing to do is to store this properties file in the same package as your `Extension` implementation, in a file named `LocalDescriptions.properties`.

11.6.5 AttributeDefinition Class

The `AttributeDefinition` class is used to create the static definition of one of a managed resource's attributes. It's a bit poorly named though, because the same interface is used to define the details of parameters to operations, and to define fields in the result of of operations.

The definition includes all the static information about the attribute/operation parameter/result field, e.g. the DMR `ModelType` of its value, whether its presence is required, whether it supports expressions, etc. See Description of the Management Model for a description of the metadata available. Almost all of this comes from the `AttributeDefinition`.

Besides basic metadata, the `AttributeDefinition` can also hold custom logic the kernel should use when dealing with the attribute/operation parameter/result field. For example, a `ParameterValidator` to use to perform special validation of values (beyond basic things like DMR type checks and defined/undefined checks), or an `AttributeParser` or `AttributeMarshaller` to use to perform customized parsing from and marshaling to XML.

WildFly Core's `controller` module provides a number of subclasses of `AttributeDefinition` used for the usual kinds of attributes. For each there is an associated builder class which you should use to build the `AttributeDefinition`. Most commonly used are `SimpleAttributeDefinition`, built by the associated `SimpleAttributeDefinitionBuilder`. This is used for attributes whose values are analogous to java primitives, `String` or byte[]. For collections, there are various subclasses of `ListAttributeDefinition` and `MapAttributeDefinition`. All have a `Builder` inner class. For complex attributes, i.e. those with a fixed set of fully defined fields, use `ObjectTypeAttributeDefinition`. (Each field in the complex type is itself specified by an `AttributeDefinition`.) Finally there's `ObjectListAttributeDefinition` and `ObjectMapAttributeDefinition` for lists whose elements are complex types and maps whose values are complex types respectively.

Here's an example of creating a simple attribute definition with extra validation of the range of allowed values:

```
static final AttributeDefinition QUEUE_LENGTH = new
SimpleAttributeDefinitionBuilder("queue-length", ModelType.INT)
                .setRequired(true)
                .setAllowExpression(true)
                .setValidator(new IntRangeValidator(1, Integer.MAX_VALUE))
                .setRestartAllServices() // means modification after resource add puts the
server in reload-required
                .build();
```

Via a bit of dark magic, the kernel knows that the `IntRangeValidator` defined here is a reliable source of information on min and max values for the attribute, so when creating the `read-resource-description` output for the attribute it will use it and output `min` and `max` metadata. For STRING attributes, `StringLengthValidator` can also be used, and the kernel will see this and provide `min-length` and `max-length` metadata. In both cases the kernel is checking for the presence of a `MinMaxValidator` and if found it provides the appropriate metadata based on the type of the attribute.

Use `EnumValidator` to restrict a STRING attribute's values to a set of legal values:

```
static final SimpleAttributeDefinition TIME_UNIT = new SimpleAttributeDefinitionBuilder("unit",
ModelType.STRING)
            .setRequired(true)
            .setAllowExpression(true)
            .setValidator(new EnumValidator<TimeUnit>(TimeUnit.class))
            .build();
```

`EnumValidator` is an implementation of `AllowedValuesValidator` that works with Java enums. You can use other implementations or write your own to do other types of restriction to certain values.

Via a bit of dark magic similar to what is done with `MinMaxValidator`, the kernel recognizes the presence of an `AllowedValuesValidator` and uses it to seed the `allowed-values` metadata in `read-resource-description` output.

Key Uses of AttributeDefinition

Your `AttributeDefinition` instances will be some of the most commonly used objects in your extension code. Following are the most typical uses. In each of these examples assume there is a `SimpleAttributeDefinition` stored in a constant FOO_AD that is available to the code. Typically FOO_AD would be a constant in the relevant `ResourceDefinition` implementation class. Assume FOO_AD represents an INT attribute.

Note that for all of these cases except for "Use in Extracting Data from the Configuration Model for Use in Runtime Services" there may be utility code that handles this for you. For example `PersistentResourceXMLParser` can handle the XML cases, and `AbstractAddStepHandler` can handle the "Use in Storing Data Provided by the User to the Configuration Model" case.

Use in XML Parsing

Here we have your extension's implementation of `XMLElementReader<List<ModelNode>>` that is being used to parse the xml for your subsystem and add `ModelNode` operations to the list that will be used to boot the server.

```
@Override
    public void readElement(final XMLExtendedStreamReader reader, final List<ModelNode>
operationList) throws XMLStreamException {
        // Create a node for the op to add our subsystem
        ModelNode addOp = new ModelNode();
        addOp.get("address").add("subsystem", "mysubsystem");
        addOp.get("operation").set("add");
        operationList.add(addOp);

        for (int i = 0; i < reader.getAttributeCount(); i++) {
            final String value = reader.getAttributeValue(i);
            final String attribute = reader.getAttributeLocalName(i);
            if (FOO_AD.getXmlName().equals(attribute) {
                FOO_AD.parseAndSetParameter(value, addOp, reader);
            } else ....
        }

        ... more parsing
    }
```

Note that the parsing code has deliberately been abbreviated. The key point is the `parseAndSetParameter` call. FOO_AD will validate the `value` read from XML, throwing an XMLStreamException with a useful message if invalid, including a reference to the current location of the `reader`. If valid, `value` will be converted to a DMR `ModelNode` of the appropriate type and stored as a parameter field of `addOp`. The name of the parameter will be what `FOO_AD.getName()` returns.

If you use `PersistentResourceXMLParser` this parsing logic is handled for you and you don't need to write it yourself.

Use in Storing Data Provided by the User to the Configuration Model

Here we illustrate code in an `OperationStepHandler` that extracts a value from a user-provided `operation` and stores it in the internal model:

```
@Override
    public void execute(OperationContext context, ModelNode operation) throws
OperationFailedException {
        // Get the Resource targeted by this operation
        Resource resource = context.readResourceForUpdate(PathAddress.EMPTY_ADDRESS);
        ModelNode model = resource.getModel();
        // Store the value of any 'foo' param to the model's 'foo' attribute
        FOO_AD.validateAndSet(operation, model);

        ... do other stuff
    }
```

As the name implies `validateAndSet` will validate the value in `operation` before setting it. A validation failure will result in an `OperationFailedException` with an appropriate message, which the kernel will use to provide a failure response to the user.

Note that `validateAndSet` will not perform expression resolution. Expression resolution is not appropriate at this stage, when we are just trying to store data to the persistent configuration model. However, it will check for expressions and fail validation if found and FOO_AD wasn't built with `setAllowExpressions(true)`.

This work of storing data to the configuration model is usually done in handlers for the `add` and `write-attribute` operations. If you base your handler implementations on the standard classes provided by WildFly Core, this part of the work will be handled for you.

Use in Extracting Data from the Configuration Model for Use in Runtime Services

This is the example you are most likely to use in your code, as this is where data needs to be extracted from the configuration model and passed to your runtime services. What your services need is custom, so there's no utility code we provide.

Assume as part of `... do other stuff` in the last example that your handler adds a step to do further work once operation execution proceeds to RUNTIME state (see Operation Execution and the `OperationContext` for more on what this means):

```
context.addStep(new OperationStepHandler() {
            @Override
            public void execute(OperationContext context, ModelNode operation) throws
OperationFailedException {

                // Get the Resource targetted by this operation
                Resource resource = context.readResource(PathAddress.EMPTY_ADDRESS);
                ModelNode model = resource.getModel();
                // Extract the value of the 'foo' attribute from the model
                int foo = FOO_AD.resolveModelAttribute(context, model).asInt();

                Service<XyZ> service = new MyService(foo);

                ... do other stuff, like install 'service' with MSC
    }
        }, Stage.RUNTIME);
```

Use `resolveModelAttribute` to extract data from the model. It does a number of things:

- reads the value from the model
- if it's an expression and expressions are supported, resolves it
- if it's undefined and undefined is allowed but FOO_AD was configured with a default value, uses the default value
- validates the result of that (which is how we check that expressions resolve to legal values), throwing OperationFailedException with a useful message if invalid
- returns that as a `ModelNode`

If when you built FOO_AD you configured it such that the user must provide a value, or if you configured it with a default value, then you know the return value of `resolveModelAttribute` will be a defined `ModelNode`. Hence you can safely perform type conversions with it, as we do in the example above with the call to `asInt()`. If FOO_AD was configured such that it's possible that the attribute won't have a defined value, you need to guard against that, e.g.:

```
ModelNode node = FOO_AD.resolveModelAttribute(context, model);
    Integer foo = node.isDefined() ? node.asInt() : null;
```

Use in Marshaling Configuration Model Data to XML

Your `Extension` must register an `XMLElementWriter<SubsystemMarshallingContext>` for each subsystem. This is used to marshal the subsystem's configuration to XML. If you don't use `PersistentResourceXMLParser` for this you'll need to write your own marshaling code, and `AttributeDefinition` will be used.

```
@Override
    public void writeContent(XMLExtendedStreamWriter writer, SubsystemMarshallingContext
context) throws XMLStreamException {
        context.startSubsystemElement(Namespace.CURRENT.getUriString(), false);

        ModelNode subsystemModel = context.getModelNode();
        // we persist foo as an xml attribute
        FOO_AD.marshalAsAttribute(subsystemModel, writer);
        // We also have a different attribute that we marshal as an element
        BAR_AD.marshalAsElement(subsystemModel, writer);
    }
```

The `SubsystemMarshallingContext` provides a `ModelNode` that represents the entire resource tree for the subsystem (including child resources). Your `XMLElementWriter` should walk through that model, using `marshalAsAttribute` or `marshalAsElement` to write the attributes in each resource. If the model includes child node trees that represent child resources, create child xml elements for those and continue down the tree.

11.6.6 OperationDefinition and OperationStepHandler Interfaces

`OperationDefinition` defines an operation, particularly its name, its parameters and the details of any result value, with `AttributeDefinition` instances used to define the parameters and result details. The `OperationDefinition` is used to generate the `read-operation-description` output for the operation, and in some cases is also used by the kernel to decide details as to how to execute the operation.

Typically `SimpleOperationDefinitionBuilder` is used to create an `OperationDefinition`. Usually you only need to create an `OperationDefinition` for custom operations. For the common `add` and `remove` operations, if you provide minimal information about your handlers to your `SimpleResourceDefinition` implementation via the `Parameters` object passed to its constructor, then `SimpleResourceDefinition` can generate a correct `OperationDefinition` for those operations.

The `OperationStepHandler` is what contains the actual logic for doing what the user requests when they invoke an operation. As its name implies, each OSH is responsible for doing one step in the overall sequence of things necessary to give effect to what the user requested. One of the things an OSH can do is add other steps, with the result that an overall operation can involve a great number of OSHs executing. (See Operation Execution and the `OperationContext` for more on this.)

Each OSH is provided in its `execute` method with a reference to the `OperationContext` that is controlling the overall operation, plus an `operation` `ModelNode` that represents the operation that particular OSH is being asked to deal with. The `operation` node will be of `ModelType.OBJECT` with the following key/value pairs:

- a key named `operation` with a value of `ModelType.STRING` that represents the name of the operation. Typically an OSH doesn't care about this information as it is written for an operation with a particular name and will only be invoked for that operation.
- a key named `address` with a value of `ModelType.LIST` with list elements of `ModelType.PROPERTY`. This value represents the address of the resource the operation targets. If this key is not present or the value is undefined or an empty list, the target is the root resource. Typically an OSH doesn't care about this information as it can more efficiently get the address from the `OperationContext` via its `getCurrentAddress()` method.
- other key/value pairs that represent parameters to the operation, with the key the name of the parameter. This is the main information an OSH would want from the `operation` node.

There are a variety of situations where extension code will instantiate an `OperationStepHandler`

- When registering a writable attribute with a `ManagementResourceRegistration` (typically in an implementation of `ResourceDefinition.registerAttributes`), an OSH must be provided to handle the `write-attribute` operation.
- When registering a read-only or read-write attribute that needs special handling of the `read-attribute` operation, an OSH must be provided.
- When registering a metric attribute, an OSH must be provided to handle the `read-attribute` operation.
- Most resources need OSHs created for the `add` and `remove` operations. These are passed to the `Parameters` object given to the `SimpleResourceDefinition` constructor, for use by the `SimpleResourceDefinition` in its implementation of the `registerOperations` method.
- If your resource has custom operations, you will instantiate them to register with a `ManagementResourceRegistration`, typically in an implementation of `ResourceDefinition.registerOperations`
- If an OSH needs to tell the `OperationContext` to add additional steps to do further handling, the OSH will create another OSH to execute that step. This second OSH is typically an inner class of the first OSH.

11.6.7 Operation Execution and the OperationContext

When the `ModelController` at the heart of the WildFly Core management layer handles a request to execute an operation, it instantiates an implementation of the `OperationContext` interface to do the work. The `OperationContext` is configured with an initial list of operation steps it must execute. This is done in one of two ways:

- During boot, multiple steps are configured, one for each operation in the list generated by the parser of the xml configuration file. For each operation, the `ModelController` finds the `ManagementResourceRegistration` that matches the address of the operation and finds the `OperationStepHandler` registered with that MRR for the operation's name. A step is added to the `OperationContext` for each operation by providing the operation `ModelNode` itself, plus the `OperationStepHandler`.
- After boot, any management request involves only a single operation, so only a single step is added. (Note that a `composite` operation is still a single operation; it's just one that internally executes via multiple steps.)

The `ModelController` then asks the `OperationContext` to execute the operation.

The `OperationContext` acts as both the engine for operation execution, and as the interface provided to `OperationStepHandler` implementations to let them interact with the rest of the system.

Execution Process

Operation execution proceeds via execution by the `OperationContext` of a series of "steps" with an `OperationStepHandler` doing the key work for each step. As mentioned above, during boot the OC is initially configured with a number of steps, but post boot operations involve only a single step initially. But even a post-boot operation can end up involving numerous steps before completion. In the case of a `/:read-resource(recursive=true)` operation, thousands of steps might execute. This is possible because one of the key things an `OperationStepHandler` can do is ask the `OperationContext` to add additional steps to execute later.

Execution proceeds via a series of "stages", with a queue of steps maintained for each stage. An `OperationStepHandler` can tell the `OperationContext` to add a step for any stage equal to or later than the currently executing stage. The instruction can either be to add the step to the head of the queue for the stage or to place it at the end of the stage's queue.

Execution of a stage continues until there are no longer any steps in the stage's queue. Then an internal transition task can execute, and the processing of the next stage's steps begins.

Here is some brief information about each stage:

Stage.MODEL

This stage is concerned with interacting with the persistent configuration model, either making changes to it or reading information from it. Handlers for this stage should not make changes to the runtime, and handlers running after this stage should not make changes to the persistent configuration model.

If any step fails during this stage, the operation will automatically roll back. Rollback of MODEL stage failures cannot be turned off. Rollback during boot results in abort of the process start.

The initial step or steps added to the `OperationContext` by the `ModelController` all execute in Stage.MODEL. This means that all `OperationStepHandler` instances your extension registers with a `ManagementResourceRegistration` must be designed for execution in `Stage.MODEL`. If you need work done in later stages your `Stage.MODEL` handler must add a step for that work.

When this stage completes, the `OperationContext` internally performs model validation work before proceeding on to the next stage. Validation failures will result in rollback.

Stage.RUNTIME

This stage is concerned with interacting with the server runtime, either reading from it or modifying it (e.g. installing or removing services or updating their configuration.) By the time this stage begins, all model changes are complete and model validity has been checked. So typically handlers in this stage read their inputs from the model, not from the original `operation` `ModelNode` provided by the user.

Most `OperationStepHandler` logic written by extension authors will be for Stage.RUNTIME. The vast majority of Stage.MODEL handling can best be performed by the base handler classes WildFly Core provides in its `controller` module. (See below for more on those.)

During boot failures in `Stage.RUNTIME` will not trigger rollback and abort of the server boot. After boot, by default failures here will trigger rollback, but users can prevent that by using the `rollback-on-runtime-failure` header. However, a RuntimeException thrown by a handler will trigger rollback.

At the end of `Stage.RUNTIME`, the `OperationContext` blocks waiting for the MSC service container to stabilize (i.e. for all services to have reached a rest state) before moving on to the next stage.

Stage.VERIFY

Service container verification work is performed in this stage, checking that any MSC changes made in `Stage.RUNTIME` had the expected effect. Typically extension authors do not add any steps in this stage, as the steps automatically added by the `OperationContext` itself are all that are needed. You can add a step here though if you have an unusual use case where you need to verify something after MSC has stabilized.

Handlers in this stage should not make any further runtime changes; their purpose is simply to do verification work and fail the operation if verification is unsuccessful.

During boot failures in `Stage.VERIFY` will not trigger rollback and abort of the server boot. After boot, by default failures here will trigger rollback, but users can prevent that by using the `rollback-on-runtime-failure` header. However, a RuntimeException thrown by a handler will trigger rollback.

There is no special transition work at the end of this stage.

Stage.DOMAIN

Extension authors should not add steps in this stage; it is only for use by the kernel.

Steps needed to execute rollout across the domain of an operation that affects multiple processes in a managed domain run here. This stage is only run on Host Contoller processes, never on servers.

Stage.DONE and ResultHandler / RollbackHandler Execution

This stage doesn't maintain a queue of steps; no `OperationStepHandler` executes here. What does happen here is persistence of any configuration changes to the xml file and commit or rollback of changes affecting multiple processes in a managed domain.

While no `OperationStepHandler` executes in this stage, following persistence and transaction commit all `ResultHandler` or `RollbackHandler` callbacks registered with the `OperationContext` by the steps that executed are invoked. This is done in the reverse order of step execution, so the callback for the last step to run is the first to be executed. The most common thing for a callback to do is to respond to a rollback by doing whatever is necessary to reverse changes made in `Stage.RUNTIME`. (No reversal of `Stage.MODEL` changes is needed, because if an operation rolls back the updated model produced by the operation is simply never published and is discarded.)

Tips About Adding Steps

Here are some useful tips about how to add steps:

- Add a step to the head of the current stage's queue if you want it to execute next, prior to any other steps. Typically you would use this technique if you are trying to decompose some complex work into pieces, with reusable logic handling each piece. There would be an `OperationStepHandler` for each part of the work, added to the head of the queue in the correct sequence. This would be a pretty advanced use case for an extension author but is quite common in the handlers provided by the kernel.
- Add a step to the end of the queue if either you don't care when it executes or if you do care and want to be sure it executes after any already registered steps.
    - A very common example of this is a `Stage.MODEL` handler adding a step for its associated `Stage.RUNTIME` work. If there are multiple model steps that will execute (e.g. at boot or as part of handling a `composite`), each will want to add a runtime step, and likely the best order for those runtime steps is the same as the order of the model steps. So if each adds its runtime step at the end, the desired result will be achieved.
    - A more sophisticated but important scenario is when a step may or may not be executing as part of a larger set of steps, i.e. it may be one step in a `composite` or it may not. There is no way for the handler to know. But it can assume that if it is part of a composite, the steps for the other operations in the composite **are already registered in the queue**. (The handler for the `composite` op guarantees this.) So, if it wants to do some work (say validation of the relationship between different attributes or resources) the input to which may be affected by possible other already registered steps, instead of doing that work itself, it should register a different step at the **end** of the queue and have that step do the work. This will ensure that when the validation step runs, the other steps in the `composite` will have had a chance to do their work. **Rule of thumb: always doing any extra validation work in an added step.**

Passing Data to an Added Step

Often a handler author will want to share state between the handler for a step it adds and the handler that added it. There are a number of ways this can be done:

- Very often the `OperationStepHandler` for the added class is an inner class of the handler that adds it. So here sharing state is easily done using final variables in the outer class.
- The handler for the added step can accept values passed to its constructor which can serve as shared state.
- The `OperationContext` includes an Attachment API which allows arbitary data to be attached to the context and retrieved by any handler that has access to the attachment key.
- The `OperationContext.addStep` methods include overloaded variants where the caller can pass in an `operation` `ModelNode` that will in turn be passed to the `execute` method of the handler for the added step. So, state can be passed via this `ModelNode`. It's important to remember though that the `address` field of the `operation` will govern what the `OperationContext` sees as the target of operation when that added step's handler executes.

Controlling Output from an Added Step

When an `OperationStepHandler` wants to report an operation result, it calls the `OperationContext.getResult()` method and manipulates the returned `ModelNode`. Similarly for failure messages it can call `OperationContext.getFailureDescription()`. The usual assumption when such a call is made is that the result or failure description being modified is the one at the root of the response to the end user. But this is not necessarily the case.

When an `OperationStepHandler` adds a step it can use one of the overloaded `OperationContext.addStep` variants that takes a `response` `ModelNode` parameter. If it does, whatever `ModelNode` it passes in will be what is updated as a result of `OperationContext.getResult()` and `OperationContext.getFailureDescription()` calls by the step's handler. This node does not need to be one that is directly associated with the response to the user.

How then does the handler that adds a step in this manner make use of whatever results the added step produces, since the added step will not run until the adding step completes execution? There are a couple of ways this can be done.

The first is to add yet another step, and provide it a reference to the `response` node used by the second step. It will execute after the second step and can read its response and use it in formulating its own response.

The second way involves using a `ResultHandler`. The `ResultHandler` for a step will execute **after** any step that it adds executes. And, it is legal for a `ResultHandler` to manipulate the "result" value for an operation, or its "failure-description" in case of failure. So, the handler that adds a step can provide to its `ResultHandler` a reference to the `response` node it passed to `addStep`, and the `ResultHandler` can in turn and use its contents to manipulate its own response.

This kind of handling wouldn't commonly be done by extension authors and great care needs to be taken if it is done. It is often done in some of the kernel handlers.

OperationStepHandler use of the OperationContext

All useful work an `OperationStepHandler` performs is done by invoking methods on the `OperationContext`. The `OperationContext` interface is extensively javadoced, so this section will just provide a brief partial overview. The OSH can use the `OperationContext` to:

- Learn about the environment in which it is executing (`getProcessType`, `getRunningMode`, `isBooting`, `getCurrentStage`, `getCallEnvironment`, `getSecurityIdentity`, `isDefaultRequiresRuntime`, `isNormalServer`)
- Learn about the operation (`getCurrentAddress`, `getCurrentAddressValue`, `getAttachmentStream`, `getAttachmentStreamCount`)
- Read the `Resource` tree (`readResource`, `readResourceFromRoot`, `getOriginalRootResource`)
- Manipulate the `Resource` tree (`createResource`, `addResource`, `readResourceForUpdate`, `removeResource`)
- Read the resource type information (`getResourceRegistration`, `getRootResourceRegistration`)
- Manipulate the resource type information (`getResourceRegistrationForUpdate`)
- Read the MSC service container (`getServiceRegistry(false)`)
- Manipulate the MSC service container (`getServiceTarget`, `getServiceRegistry(true)`, `removeService`)
- Manipulate the process state (`reloadRequired`, `revertReloadRequired`, `restartRequired`, `revertRestartRequired`
- Resolve expressions (`resolveExpressions`)
- Manipulate the operation response (`getResult`, `getFailureDescription`, `attachResultStream`, `runtimeUpdateSkipped`)
- Force operation rollback (`setRollbackOnly`)
- Add other steps (`addStep`)
- Share data with other steps (`attach`, `attachIfAbsent`, `getAttachment`, `detach`)
- Work with capabilities (numerous methods)
- Emit notifications (`emit`)
- Request a callback to a `ResultHandler` or `RollbackHandler` (`completeStep`)

Locking and Change Visibility

The `ModelController` and `OperationContext` work together to ensure that only one operation at a time is modifying the state of the system. This is done via an exclusive lock maintained by the `ModelController`. Any operation that does not need to write never requests the lock and is able to proceed without being blocked by an operation that holds the lock (i.e. writes do not block reads.) If two operations wish to concurrently write, one or the other will get the lock and the loser will block waiting for the winner to complete and release the lock.

The `OperationContext` requests the exclusive lock the first time any of the following occur:

- A step calls one of its methods that indicates a wish to modify the resource tree (`createResource`, `addResource`, `readResourceForUpdate`, `removeResource`)
- A step calls one of its methods that indicates a wish to modify the `ManagementResourceRegistration` tree (`getResourceRegistrationForUpdate`)
- A step calls one of its methods that indicates a desire to change MSC services (`getServiceTarget`, `removeService` or `getServiceRegistry` with the `modify` param set to `true`)
- A step calls one of its methods that manipulates the capability registry (various)
- A step explicitly requests the lock by calling the `acquireControllerLock` method (doing this is discouraged)

The step that acquired the lock is tracked, and the lock is released when the `ResultHandler` added by that step has executed. (If the step doesn't add a result handler, a default no-op one is automatically added).

When an operation first expresses a desire to manipulate the `Resource` tree or the capability registry, a private copy of the tree or registry is created and thereafter the `OperationContext` works with that copy. The copy is published back to the `ModelController` in `Stage.DONE` if the operation commits. Until that happens any changes to the tree or capability registry made by the operation are invisible to other threads. If the operation does not commit, the private copies are simply discarded.

However, the `OperationContext` does not make a private copy of the `ManagementResourceRegistration` tree before manipulating it, nor is there a private copy of the MSC service container. So, any changes made by an operation to either of those are immediately visible to other threads.

11.6.8 Resource Interface

An instance of the `Resource` interface holds the state for a particular instance of a type defined by a `ManagementResourceRegistration`. Referring back to the analogy mentioned earlier the `ManagementResourceRegistration` is analogous to a Java class while the `Resource` is analogous to an instance of that class.

The `Resource` makes available state information, primarily

- Some descriptive metadata, such as its address, whether it is runtime-only and whether it represents a proxy to a another primary resource that resides on another process in a managed domain
- A `ModelNode` of `ModelType.OBJECT` whose keys are the resource's attributes and whose values are the attribute values
- Links to child resources such that the resources form a tree

Creating Resources

Typically extensions create resources via `OperationStepHandler` calls to the `OperationContext.createResource` method. However it is allowed for handlers to use their own `Resource` implementations by instantiating the resource and invoking `OperationContext.addResource`. The `AbstractModelResource` class can be used as a base class.

Runtime-Only and Synthetic Resources and the PlaceholderResourceEntry Class

A runtime-only resource is one whose state is not persisted to the xml configuration file. Many runtime-only resources are also "synthetic" meaning they are not added or removed as a result of user initiated management operations. Rather these resources are "synthesized" in order to allow users to use the management API to examine some aspect of the internal state of the process. A good example of synthetic resources are the resources in the `/core-service=platform-mbeans` branch of the resource tree. There are resources there that represent various aspects of the JVM (classloaders, memory pools, etc) but which resources are present entirely depends on what the JVM is doing, not on any management action. Another example are resources representing "core queues" in the WildFly messaging and messaging-artemismq subsystems. Queues are created as a result of activity in the message broker which may not involve calls to the management API. But for each such queue a management resource is available to allow management users to perform management operations against the queue.

It is a requirement of execution of a management operation that the `OperationContext` can navigate through the resource tree to a `Resource` object located at the address specified. This requirement holds true even for synthetic resources. How can this be handled, given the fact these resources are not created in response to management operations?

The trick involves using special implementations of `Resource`. Let's imagine a simple case where we have a parent resource which is fairly normal (i.e. it holds persistent configuration and is added via a user's `add` operation) except for the fact that one of its child types represents synthetic resources (e.g. message queues). How would this be handled?

First, the parent resource would require a custom implementation of the `Resource` interface. The `OperationStepHandler` for the `add` operation would instantiate it, providing it with access to whatever API is needed for it to work out what items exist for which a synthetic resource should be made available (e.g. an API provided by the message broker that provides access to its queues). The `add` handler would use the `OperationContext.addResource` method to tie this custom resource into the overall resource tree.

The custom `Resource` implementation would use special implementations of the various methods that relate to accessing children. For all calls that relate to the synthetic child type (e.g. core-queue) the custom implementation would use whatever API call is needed to provide the correct data for that child type (e.g. ask the message broker for the names of queues).

A nice strategy for creating such a custom resource is to use delegation. Use `Resource.Factory.create}()` to create a standard resource. Then pass it to the constructor of your custom resource type for use as a delegate. The custom resource type's logic is focused on the synthetic children; all other work it passes on to the delegate.

What about the synthetic resources themselves, i.e. the leaf nodes in this part of the tree? These are created on the fly by the parent resource in response to `getChild`, `requireChild`, `getChildren` and `navigate` calls that target the synthetic resource type. These created-on-the-fly resources can be very lightweight, since they store no configuration model and have no children. The `PlaceholderResourceEntry` class is perfect for this. It's a very lightweight `Resource` implementation with minimal logic that only stores the final element of the resource's address as state.

See `LoggingResource` in the WildFly Core logging subsystem for an example of this kind of thing. Searching for other uses of `PlaceholderResourceEntry` will show other examples.

11.6.9 DeploymentUnitProcessor Interface

TODO

11.6.10 Useful classes for implementing OperationStepHandler

The WildFly Core `controller` module includes a number of `OperationStepHandler` implementations that in some cases you can use directly, and that in other cases can serve as the base class for your own handler implementation. In all of these a general goal is to eliminate the need for your code to do anything in `Stage.MODEL` while providing support for whatever is appropriate for `Stage.RUNTIME`.

Add Handlers

`AbstractAddStepHandler` is a base class for handlers for `add` operations. There are a number of ways you can configure its behavior, the most commonly used of which are to:

- Configure its behavior in `Stage.MODEL` by passing to its constructor `AttributeDefinition` and `RuntimeCapability` instances for the attributes and capabilities provided by the resource. The handler will automatically validate the operation parameters whose names match the provided attributes and store their values in the model of the newly added `Resource`. It will also record the presence of the given capabilities.
- Control whether a `Stage.RUNTIME` step for the operation needs to be added, by overriding the `protected boolean requiresRuntime(OperationContext context)` method. Doing this is atypical; the standard behavior in the base class is appropriate for most cases.
- Implement the primary logic of the `Stage.RUNTIME` step by overriding the `protected void performRuntime(final OperationContext context, final ModelNode operation, final Resource resource)` method. This is typically the bulk of the code in an `AbstractAddStepHandler` subclass. This is where you read data from the `Resource` model and use it to do things like configure and install MSC services.
- Handle any unusual needs of any rollback of the `Stage.RUNTIME` step by overriding `protected void rollbackRuntime(OperationContext context, final ModelNode operation, final Resource resource)`. Doing this is not typically needed, since if the rollback behavior needed is simply to remove any MSC services installed in `performRuntime`, the `OperationContext` will do this for you automatically.

`AbstractBoottimeAddStepHandler` is a subclass of `AbstractAddStepHandler` meant for use by `add` operations that should only do their normal `Stage.RUNTIME` work in server, boot, with the server being put in `reload-required` if executed later. Primarily this is used for `add` operations that register `DeploymentUnitProcessor` implementations, as this can only be done at boot.

Usage of `AbstractBoottimeAddStepHandler` is the same as for `AbstractAddStepHandler` except that instead of overriding `performRuntime` you override `protected void performBoottime(OperationContext context, ModelNode operation, Resource resource)`.

A typical thing to do in `performBoottime` is to add a special step that registers one or more `DeploymentUnitProcessor`s.

```
@Override
    public void performBoottime(OperationContext context, ModelNode operation, final Resource
resource)
            throws OperationFailedException {

        context.addStep(new AbstractDeploymentChainStep() {
            @Override
            protected void execute(DeploymentProcessorTarget processorTarget) {


processorTarget.addDeploymentProcessor(RequestControllerExtension.SUBSYSTEM_NAME,
Phase.STRUCTURE, Phase.STRUCTURE_GLOBAL_REQUEST_CONTROLLER, new
RequestControllerDeploymentUnitProcessor());
            }
        }, OperationContext.Stage.RUNTIME);

        ... do other things
```

Remove Handlers

TODO `AbstractRemoveStepHandler` `ServiceRemoveStepHandler`

Write attribute handlers

TODO `AbstractWriteAttributeHandler`

Reload-required handlers

`ReloadRequiredAddStepHandler` `ReloadRequiredRemoveStepHandler` `ReloadRequiredWriteAttributeHandler`

Use these for cases where, post-boot, the change to the configuration model made by the operation cannot be reflected in the runtime until the process is reloaded. These handle the mechanics of recording the need for reload and reverting it if the operation rolls back.

Restart Parent Resource Handlers

`RestartParentResourceAddHandler` `RestartParentResourceRemoveHandler` `RestartParentWriteAttributeHandler`

Use these in cases where a management resource doesn't directly control any runtime services, but instead simply represents a chunk of configuration that a parent resource uses to configure services it installs. (Really, this kind of situation is now considered to be a poor management API design and is discouraged. Instead of using child resources for configuration chunks, complex attributes on the parent resource should be used.)

These handlers help you deal with the mechanics of the fact that, post-boot, any change to the child resource likely requires a restart of the service provided by the parent.

Model Only Handlers

`ModelOnlyAddStepHandler` `ModelOnlyRemoveStepHandler` `ModelOnlyWriteAttributeHandler`

Use these for cases where the operation never affects the runtime, even at boot. All it does is update the configuration model. In most cases such a thing would be odd. These are primarily useful for legacy subsystems that are no longer usable on current version servers and thus will never do anything in the runtime. However, current version Domain Controllers must be able to understand the subsystem's configuration model to allow them to manage older Host Controllers running previous versions where the subsystem is still usable by servers. So these handlers allow the DC to maintain the configuration model for the subsystem.

Misc

`AbstractRuntimeOnlyHandler` is used for custom operations that don't involve the configuration model. Create a subclass and implement the `protected abstract void executeRuntimeStep(OperationContext context, ModelNode operation)` method. The superclass takes care of adding a `Stage.RUNTIME` step that calls your method.

`ReadResourceNameOperationStepHandler` is for cases where a resource type includes a 'name' attribute whose value is simply the value of the last element in the resource's address. There is no need to store the value of such an attribute in the resource's model, since it can always be determined from the resource address. But, if the value is not stored in the resource model, when the attribute is registered with `ManagementResourceRegistration.registerReadAttribute` an `OperationStepHandler` to handle the `read-attribute` operation must be provided. Use `ReadResourceNameOperationStepHandler` for this. (Note that including such an attribute in your management API is considered to be poor practice as it's just redundant data.)

11.7 CLI Extensibility for Layered Products

In addition to supporting the ServiceLoader extension mechanism to load command handlers coming from outside of the CLI codebase, starting from the wildfly-core-1.0.0.Beta1 release the CLI running in a modular classloading environment can be extended with commands exposed in server extension modules. The CLI will look for and register extension commands when it (re-)connects to the controller by iterating through the registered by that time extensions and using the ServiceLoader mechanism on the extension modules. (Note, that this mechanism will work only for extensions available in the server installation the CLI is launched from.)

Here is an example of a simple command handler and its integration.

```
package org.jboss.as.test.cli.extensions;public class ExtCommandHandler extends
org.jboss.as.cli.handlers.CommandHandlerWithHelp {

package org.jboss.as.test.cli.extensions;
public class ExtCommandHandler extends org.jboss.as.cli.handlers.CommandHandlerWithHelp {


    public static final String NAME = "ext-command";
    public static final String OUTPUT = "hello world!";


    public CliExtCommandHandler() {
        super(NAME, false);
    }


    @Override
    protected void doHandle(CommandContext ctx) throws CommandLineException {
        ctx.printLine(OUTPUT);
    }
}
```

The command will simply print a message to the terminal. The next step is to implement the CLI CommandHandlerProvider interface.

```
package org.jboss.as.test.cli.extensions;
public class ExtCommandHandlerProvider implements org.jboss.as.cli.CommandHandlerProvider {


    @Override
    public CommandHandler createCommandHandler(CommandContext ctx) {
        return new ExtCommandHandler();
    }


    /**
     * Whether the command should be available in tab-completion.
     */
    @Override
    public boolean isTabComplete() {
        return true;
    }


    /**
     * Command name(s).
     */
    @Override
    public String[] getNames() {
        return new String[]{ExtCommandHandler.NAME};
    }
}
```

The final step is to include **META-INF/services/org.jboss.as.cli.CommandHandlerProvider** entry into the JAR file containing the classes above with value **org.jboss.as.test.cli.extensions.ExtCommandHandlerProvider**.

11.8 All WildFly documentation

There are several guides in the WildFly documentation series. This list gives an overview of each of the guides:

*Getting Started Guide - Explains how to download and start WildFly.
*Getting Started Developing Applications Guide - Talks you through developing your first applications on WildFly, and introduces you to JBoss Tools and how to deploy your applications.
*JavaEE 6 Tutorial - A Java EE 6 Tutorial.
*Admin Guide - Tells you how to configure and manage your WildFly instances.
*Developer Guide - Contains concepts that you need to be aware of when developing applications for WildFly. Classloading is explained in depth.
*High Availability Guide - Reference guide for how to set up clustered WildFly instances.
*Extending WildFly - A guide to adding new functionality to WildFly.

11.9 CLI extensibility for layered products

In addition to supporting the ServiceLoader extension mechanism to load command handlers coming from outside of the CLI codebase, starting from the wildfly-core-1.0.0.Beta1 release the CLI running in a modular classloading environment can be extended with commands exposed in server extension modules. The CLI will look for and register extension commands when it (re-)connects to the controller by iterating through the registered by that time extensions and using the ServiceLoader mechanism on the extension modules. (Note, that this mechanism will work only for extensions available in the server installation the CLI is launched from.)

Here is an example of a simple command handler and its integration.

```
package org.jboss.as.test.cli.extensions;public class ExtCommandHandler extends
org.jboss.as.cli.handlers.CommandHandlerWithHelp {

package org.jboss.as.test.cli.extensions;
public class ExtCommandHandler extends org.jboss.as.cli.handlers.CommandHandlerWithHelp {


    public static final String NAME = "ext-command";
    public static final String OUTPUT = "hello world!";


    public CliExtCommandHandler() {
        super(NAME, false);
    }


    @Override
    protected void doHandle(CommandContext ctx) throws CommandLineException {
        ctx.printLine(OUTPUT);
    }
}
```

The command will simply print a message to the terminal. The next step is to implement the CLI CommandHandlerProvider interface.

```
package org.jboss.as.test.cli.extensions;
public class ExtCommandHandlerProvider implements org.jboss.as.cli.CommandHandlerProvider {


    @Override
    public CommandHandler createCommandHandler(CommandContext ctx) {
        return new ExtCommandHandler();
    }


    /**
     * Whether the command should be available in tab-completion.
     */
    @Override
    public boolean isTabComplete() {
        return true;
    }


    /**
     * Command name(s).
     */
    @Override
    public String[] getNames() {
        return new String[]{ExtCommandHandler.NAME};
    }
}
```

The final step is to include **META-INF/services/org.jboss.as.cli.CommandHandlerProvider** entry into the JAR file containing the classes above with value **org.jboss.as.test.cli.extensions.ExtCommandHandlerProvider**.

11.10 Domain Mode Subsystem Transformers

11.10.1 Abstract

A WildFly domain may consist of a new Domain Controller (DC) controlling slave Host Controllers (HC) running older versions. Each slave HC maintains a copy of the centralized domain configuration, which they use for controlling their own servers. In order for the slave HCs to understand the configuration from the DC, transformation is needed, whereby the DC translates the configuration and operations into something the slave HCs can understand.

11.10.2 Background

WildFly comes with a domain mode which allows you to have one Host Controller acting as the Domain Controller. The Domain Controller's job is to maintain the centralized domain configuration. Another term for the DC is 'Master Host Controller'. Before explaining why transformers are important and when they should be used, we will revisit how the domain configuration is used in domain mode.

The centralized domain configuration is stored in `domain.xml`. This is only ever parsed on the DC, and it has the following structure:

- `extensions` - contains:
  - `extension` - a references to a module that bootstraps the `org.jboss.as.controller.Extension` implementation used to bootstrap your subsystem parsers and initialize the resource definitions for your subsystems.
- `profiles` - contains:
  - `profile` - a named set of:
    - `subsystem` - contains the configuration for a subsystem, using the parser initialized by the subsystem's extension.
- `socket-binding-groups` - contains:
  - `socket-binding-group` - a named set of:
    - `socket-binding` - A named port on an interface which can be referenced from the `subsystem` configurations for subsystems opening sockets.
- `server-groups` - contains
  - `server-group` - this has a name and references a `profile` and a `socket-binding-group`. The HCs then reference the `server-group` name from their `<servers>` section in `host.xml`.

When the DC parses `domain.xml`, it is transformed into `add` (and in some cases `write-attribute`) operations just as explained in Parsing and marshalling of the subsystem xml. These operations build up the model on the DC.

A HC wishing to join the domain and use the DC's centralized configuration is known as a 'slave HC'. A slave HC maintains a copy of the DC's centralized domain configuration. This copy of the domain configuration is used to start its servers. This is done by asking the domain model to `describe` itself, which in turn asks the subsystems to `describe` themselves. The `describe` operation for a subsystem looks at the state of the subsystem model and produces the `add` operations necessary to create the subsystem on the server. The same mechanism also takes place on the DC (bear in mind that the DC is also a HC, which can have its own servers), although of course its copy of the domain configuration is the centralized one.

There are two steps involved in keeping the keeping the slave HC's domain configuration in sync with the centralized domain configuration.

- getting the initial domain model
- an operation changes something in the domain configuration

Let's look a bit closer at what happens in each of these steps.

Getting the initial domain model

When a slave HC connects to the DC it obtains a copy of the domain model from the DC. This is done in a simpler serialized format, different from the operations that built up the model on the DC, or the operations resulting from the `describe` step used to bootstrap the servers. They describe each address that exists in the DC's model, and contain the attributes set for the resource at that address. This serialized form looks like this:

```
[{
    "domain-resource-address" => [],
    "domain-resource-model" => {
        "management-major-version" => 2,
        "management-minor-version" => 0,
        "management-micro-version" => 0,
        "release-version" => "8.0.0.Beta1-SNAPSHOT",
        "release-codename" => "WildFly"
    }
},
{
    "domain-resource-address" => [("extension" => "org.jboss.as.clustering.infinispan")],
    "domain-resource-model" => {"module" => "org.jboss.as.clustering.infinispan"}
},
--SNIP - the rest of the extensions --
{
    "domain-resource-address" => [("extension" => "org.jboss.as.weld")],
    "domain-resource-model" => {"module" => "org.jboss.as.weld"}
},
{
    "domain-resource-address" => [("system-property" => "java.net.preferIPv4Stack")],
    "domain-resource-model" => {
        "value" => "true",
        "boot-time" => undefined
    }
},
{
    "domain-resource-address" => [("profile" => "full-ha")],
    "domain-resource-model" => undefined
},
{
    "domain-resource-address" => [
        ("profile" => "full-ha"),
        ("subsystem" => "logging")
    ],
    "domain-resource-model" => {}
},
{
    "domain-resource-address" => [sss|WFLY8:Example subsystem],
    "domain-resource-model" => {
        "level" => "INFO",
        "enabled" => undefined,
        "encoding" => undefined,
        "formatter" => "%d{HH:mm:ss,SSS} %-5p [%c] (%t) %s%E%n",
        "filter-spec" => undefined,
        "autoflush" => undefined,
        "target" => undefined,
        "named-formatter" => undefined
    }
},
--SNIP---
```

The slave HC then applies these one at a time and builds up the initial domain model. It needs to do this before it can start any of its servers.

An operation changes something in the domain configuration

Once a domain is up and running we can still change things in the domain configuration. These changes must happen when connected to the DC, and are then propagated to the slave HCs, which then in turn propagate the changes to any servers running in a server group affected by the changes made. In this example:

```
[disconnected /] connect
[domain@localhost:9990 /]
/profile=full/subsystem=datasources/data-source=ExampleDS:write-attribute(name=enabled,value=false
"outcome" => "success",
    "result" => undefined,
    "server-groups" => {"main-server-group" => {"host" => {
        "slave" => {"server-one" => {"response" => {
            "outcome" => "success",
            "result" => undefined,
            "response-headers" => {
                "operation-requires-restart" => true,
                "process-state" => "restart-required"
            }
        }}},
        "master" => {
            "server-one" => {"response" => {
                "outcome" => "success",
                "response-headers" => {
                    "operation-requires-restart" => true,
                    "process-state" => "restart-required"
                }
            }},
            "server-two" => {"response" => {
                "outcome" => "success",
                "response-headers" => {
                    "operation-requires-restart" => true,
                    "process-state" => "restart-required"
                }
            }}
        }
    }}}
}
```

the DC propagates the changes to itself `host=master`, which in turn propagates it to its two servers belonging to `main-server-group` which uses the `full` profile. More interestingly, it also propagates it to `host=slave` which updates its local copy of the domain model, and then propagates the change to its `server-one` which belongs to `main-server-group` which uses the `full` profile.

11.10.3 Versions and backward compatibility

A HC and its servers will always be the same version of WildFly (they use the same module path and jars). However, the DC and the slave HCs do not necessarily need to be the same version. One of the points in the original specification for WildFly is that

Important

A Domain Controller should be able to manage slave Host Controllers older than itself.

This means that for example a WildFly 10.1 DC should be able to work with slave HCs running WildFly 10. The opposite is not true, the DC must be the same or the newest version in the domain.

Versioning of subsystems

To help with being able to know what is compatible we have versions within the subsystems, this is stored in the subsystem's extension. When registering the subsystem you will typically see something like:

```
public class SomeExtension implements Extension {

    private static final String SUBSYSTEM_NAME = "my-subsystem"'

    private static final int MANAGEMENT_API_MAJOR_VERSION = 2;
    private static final int MANAGEMENT_API_MINOR_VERSION = 0;
    private static final int MANAGEMENT_API_MICRO_VERSION = 0;

    /**
     * {@inheritDoc}
     * @see
org.jboss.as.controller.Extension#initialize(org.jboss.as.controller.ExtensionContext)
     */
    @Override
    public void initialize(ExtensionContext context) {

        // IMPORTANT: Management API version != xsd version! Not all Management API changes
result in XSD changes
        SubsystemRegistration registration = context.registerSubsystem(SUBSYSTEM_NAME,
MANAGEMENT_API_MAJOR_VERSION,
                MANAGEMENT_API_MINOR_VERSION, MANAGEMENT_API_MICRO_VERSION);

        //Register the resource definitions
        ....
    }
    ....
}
```

Which sets the `ModelVersion` of the subsystem.

Important

Whenever something changes in the subsystem, such as:

- an attribute is added or removed from a resource
- a attribute is renamed in a resource
- an attribute has its type changed
- an attribute or operation parameter's nillable or allows expressions is changed
- an attribute or operation parameter's default value changes
- a child resource type is added or removed
- an operation is added or removed
- an operation has its parameters changed

and the current version of the subsystem has been part of a Final release of WildFly, we **must** bump the version of the subsystem.

Once it has been increased you can of course make more changes until the next Final release without more version bumps. It is also worth noting that a new WildFly release does not automatically mean a new version for the subsystem, the new version is only needed if something was changed. For example the `jaxrs` subsystem has remained on 1.0.0 for all versions of WildFly and JBoss AS 7.

You can find the `ModelVersion` of a subsystem by querying its extension:

```
domain@localhost:9990 /]
/extension=org.jboss.as.clustering.infinispan:read-resource(recursive=true)
{
    "outcome" => "success",
    "result" => {
        "module" => "org.jboss.as.clustering.infinispan",
        "subsystem" => {"infinispan" => {
            "management-major-version" => 2,
            "management-micro-version" => 0,
            "management-minor-version" => 0,
            "xml-namespaces" => [jboss:domain:infinispan:1.0",
                "urn:jboss:domain:infinispan:1.1",
                "urn:jboss:domain:infinispan:1.2",
                "urn:jboss:domain:infinispan:1.3",
                "urn:jboss:domain:infinispan:1.4",
                "urn:jboss:domain:infinispan:2.0"]
        }}
    }
}
```

11.10.4 The role of transformers

Now that we have mentioned the slave HCs registration process with the DC, and know about ModelVersions, it is time to mention that when registering with the DC, the slave HC will send across a list of all its subsystem ModelVersions. The DC maintains this information in a registry for each slave HC, so that it knows which transformers (if any) to invoke for a legacy slave. We will see how to write and register transformers later on in How do I write a transformer. Slave HCs from version 7.2.0 onwards will also include a list of resources that they ignore (see Ignoring resources on legacy hosts), and the DC will maintain this information in its registry. The DC will not send across any resources that it knows a slave ignores during the initial domain model transfer. When forwarding operations onto the slave HCs, the DC will skip forwarding those to slave HCs ignoring those resources.

There are two kinds of transformers:

- resource transformers
- operation transformers

The main function of transformers is to transform a subsystem to something that the legacy slave HC can understand, or to aggressively reject things that the legacy slave HC will not understand. Rejection, in this context, essentially means, that the resource or operation cannot safely be transformed to something valid on the slave, so the transformation fails. We will see later how to reject attributes in Rejecting attributes, and child resources in Reject child resource.

Both resource and operation transformers are needed, but take effect at different times. Let us use the `weld` subsystem, which is relatively simple, as an example. In JBoss AS 7.2.0 and lower it had a ModelVersion of 1.0.0, and its resource description was as follows:

```
{
                    "description" => "The configuration of the weld subsystem.",
                    "attributes" => {},
                    "operations" => {
                        "remove" => {
                            "operation-name" => "remove",
                            "description" => "Operation removing the weld subsystem.",
                            "request-properties" => {},
                            "reply-properties" => {}
                        },
                        "add" => {
                            "operation-name" => "add",
                            "description" => "Operation creating the weld subsystem.",
                            "request-properties" => {},
                            "reply-properties" => {}
                        }
                    },
                    "children" => {}
                },
```

In WildFly 8, it has a ModelVersion of 2.0.0 and has added two attributes, `require-bean-descriptor` and `non-portable` mode:

```
{
        "description" => "The configuration of the weld subsystem.",
        "attributes" => {
            "require-bean-descriptor" => {
                "type" => BOOLEAN,
                "description" => "If true then implicit bean archives without bean descriptor
file (beans.xml) are ignored by Weld",
                "expressions-allowed" => true,
                "nillable" => true,
                "default" => false,
                "access-type" => "read-write",
                "storage" => "configuration",
                "restart-required" => "no-services"
            },
            "non-portable-mode" => {
                "type" => BOOLEAN,
                "description" => "If true then the non-portable mode is enabled. The
non-portable mode is suggested by the specification to overcome problems with legacy
applications that do not use CDI SPI properly and may be rejected by more strict validation in
CDI 1.1.",
                "expressions-allowed" => true,
                "nillable" => true,
                "default" => false,
                "access-type" => "read-write",
                "storage" => "configuration",
                "restart-required" => "no-services"
            }
        },
        "operations" => {
            "remove" => {
                "operation-name" => "remove",
                "description" => "Operation removing the weld subsystem.",
                "request-properties" => {},
                "reply-properties" => {}
            },
            "add" => {
                "operation-name" => "add",
                "description" => "Operation creating the weld subsystem.",
                "request-properties" => {
                    "require-bean-descriptor" => {
                        "type" => BOOLEAN,
                        "description" => "If true then implicit bean archives without bean
descriptor file (beans.xml) are ignored by Weld",
                        "expressions-allowed" => true,
                        "required" => false,
                        "nillable" => true,
                        "default" => false
                    },
                    "non-portable-mode" => {
                        "type" => BOOLEAN,
                        "description" => "If true then the non-portable mode is enabled. The
non-portable mode is suggested by the specification to overcome problems with legacy
applications that do not use CDI SPI properly and may be rejected by more strict validation in
```

```
CDI 1.1.",
                        "expressions-allowed" => true,
                        "required" => false,
                        "nillable" => true,
                        "default" => false
                    }
                },
                "reply-properties" => {}
            }
        },
        "children" => {}
    }
```

In the rest of this section we will assume that we are running a DC running WildFly 8 so it will have ModelVersion 2.0.0 of the weld subsystem, and that we are running a slave using ModelVersion 1.0.0 of the weld subsystem.

Important

Transformation always takes place on the Domain Controller, and is done when sending across the initial domain model AND forwarding on operations to legacy slave HCs.

Resource transformers

When copying over the centralized domain configuration as mentioned in Getting the initial domain model, we need to make sure that the copy of the domain model is something that the servers running on the legacy slave HC understand. So if the centralized domain configuration had any of the two new attributes set, we would need to reject the transformation in the transformers. One reason for this is to keep things consistent, it doesn't look good if you connect to the slave HC and find attributes and/or child resources when doing `:read-resource` which are not there when you do `:read-resource-description`. Also, to make life easier for subsystem writers, most instances of the `describe` operation use a standard implementation which would include these attributes when creating the `add` operation for the server, which could cause problems there.

Another, more concrete example from the logging subsystem is that it allows a '`%K{...}`' in the pattern formatter which makes the formatter use color:

```
<pattern-formatter pattern="%K{level}%d{HH:mm:ss,SSS} %-5p [%c] (%t) %s%E%n"/>
```

This '`%K{...}`' however was introduced in JBoss AS < 7.1.3 (ModelVersion 1.2.0), so if that makes it across to a slave HC running an older version, the servers **will** fail to start up. So the logging extension registers transformers to strip out the '`%K{...}`' from the attribute value (leaving '`%-5p`

```
[%c]
```

`(%t) %s%E%n"`') so that the old slave HC's servers can understand it.

Rejection in resource transformers

Only slave HCs from JBoss AS 7.2.0 and newer inform the DC about their ignored resources (see Ignoring resources on legacy hosts). This means that if a transformer on the DC rejects transformation for a legacy slave HC, exactly what happens to the slave HC depends on the version of the slave HC. If the slave HC is:

- *older than 7.2.0* - the DC has no means of knowing if the slave HC has ignored the resource being rejected or not. So we log a warning on the DC, and send over the serialized part of that model anyway. If the slave HC has ignored the resource in question, it does not apply it. If the slave HC has not ignored the resource in question, it will apply it, but no failure will happen until it tries to start a server which references this bad configuration.
- *7.2.0 or newer* - If a resource is ignored on the slave HC, the DC knows about this, and will not attempt to transform or send the resource across to the slave HC. If the resource transformation is rejected, we know the resource was not ignored on the slave HC and so we can aggressively fail the transformation, which in turn will cause the slave HC to fail to start up.

Operation transformers

When An operation changes something in the domain configuration the operation gets sent across to the slave HCs to update their copies of the domain model. The slave HCs then forward this operation onto the affected servers. The same considerations as in Resource transformers are true, although operation transformers give you quicker 'feedback' if something is not valid. If you try to execute:

```
/profile=full/subsystem=weld:write-attribute(name=require-bean-descriptor, value=false)
```

This will fail on the legacy slave HC since its version of the subsystem does not contain any such attribute. However, it is best to aggressively reject in such cases.

Rejection in operation transformers

For transformed operations we can always know if the operation is on an ignored resource in the legacy slave HC. In 7.2.0 onwards, we know this through the DC's registry of ignored resources on the slave. In older versions of slaves, we send the operation across to the slave, which tries to invoke the operation. If the operation is against an ignored resource we inform the DC about this fact. So as part of the transformation process, if something gets rejected we can (and do!) fail the transformation aggressively. If the operation invoked on the DC results in the operation being sent across to 10 slave HCs and one of them has a legacy version which ends up rejecting the transformation, we rollback the operation across the whole domain.

Different profiles for different versions

Now for the `weld` example we have been using there is a slight twist. We have the new `require-bean-descriptor` and `non-portable-mode` attributes. These have been added in WildFly 8 which supports Java EE 7, and thus CDI 1.1. JBoss AS 7.x supports Java EE 6, and thus CDI 1.0. In CDI 1.1 the values of these attributes are tweakable, so they can be set to either `true` or `false`. The default behaviour for these in CDI 1.1, if not set, is that they are `false`. However, for CDI 1.0 these were not tweakable, and with the way the subsystem in JBoss AS 7.x worked is similar to if they are set to `true`.

The above discussion implies that to use the weld subsystem on a legacy slave HC, the `domain.xml` configuration for it must look like:

```
<subsystem xmlns="urn:jboss:domain:weld:2.0"
      require-bean-descriptor="true"
      non-portable-mode="true"/>
```

We will see the exact mechanics for how this is actually done later but in short when pushing this to a legacy slave DC we register transformers which reject the transformation if these attributes are not set to `true` since that implies some behavior not supported on the legacy slave DC. If they are `true`, all is well, and the transformers discard, or remove, these attributes since they don't exist in the legacy model. This removal is fine since they have the values which would result in the behavior assumed on the legacy slave HC.

That way the older slave HCs will work fine. However, we might also have WildFly 8 slave HCs in our domain, and they are missing out on the new features introduced by the attributes introduced in ModelVersion 2.0.0. If we do

```
<subsystem xmlns="urn:jboss:domain:weld:2.0"
      require-bean-descriptor="false"
      non-portable-mode="false"/>
```

then it will fail when doing transformation for the legacy controller. The solution is to put these in two different profiles in `domain.xml`

```
<domain>
....
  <profiles>
    <profile name="full">
      <subsystem xmlns="urn:jboss:domain:weld:2.0"
        require-bean-descriptor="false"
        non-portable-mode="false"/>
      ...
    </profile>
    <profile name="full-legacy">
      <subsystem xmlns="urn:jboss:domain:weld:2.0"
        require-bean-descriptor="true"
        non-portable-mode="true"/>
      ...
    </profile>
  </profiles>
  ...
  <server-groups>
    <server-group name="main-server-group" profile="full">
      ....
    <server-group>
    <server-group name="main-server-group-legacy" profile="full-legacy">
      ....
    <server-group>
  </server-groups>
</domain>
```

Then have the HCs using WildFly 8 make their servers reference the `main-server-group` server group, and the HCs using older versions of WildFly 8 make their servers reference the `main-server-group-legacy` server group.

Ignoring resources on legacy hosts

Booting the above configuration will still cause problems on legacy slave HCs, especially if they are JBoss AS 7.2.0 or later. The reason for this is that when they register themselves with the DC it lets the DC know which `ignored resources` they have. If the DC comes to transform something it should reject for a slave HC and it is not part of its ignored resources it will aggressively fail the transformation. Versions of JBoss AS older than 7.2.0 still have this ignored resources mechanism, but don't let the DC know about what they have ignored so the DC cannot reject aggressively - instead it will log some warnings. However, it is still good practice to ignore resources you are not interested in regardless of which legacy version the slave HC is running.

To ignore the profile we cannot understand we do the following in the legacy slave HC's `host.xml`

```
<host xmlns="urn:jboss:domain:1.3" name="slave">
...
    <domain-controller>
       <remote host="${jboss.test.host.master.address}" port="${jboss.domain.master.port:9999}"
security-realm="ManagementRealm">
            <ignored-resources type="profile">
                <instance name="full-legacy"/>
            </ignored-resources>
       </remote>
    </domain-controller>
....
</host>
```

Important

Any top-level resource type can be ignored `profile`, `extension`, `server-group` etc. Ignoring a resource instance ignores that resource, and all its children.

11.10.5 How do I know what needs to be transformed?

There is a set of related classes in the `org.wildfly.legacy.util` package to help you determine this. These now live at https://github.com/wildfly/wildfly-legacy-test/tree/master/tools/src/main/java/org/wildfly/legacy/util.
They are all runnable in your IDE, just start the WildFly or JBoss AS 7 instances as described below.

Getting data for a previous version

https://github.com/wildfly/wildfly-legacy-test/tree/master/tools/src/main/resources/legacy-models contains the output for the previous WildFly/JBoss AS 7 versions, so check if the files for the version you want to check backwards compatibility are there yet. If not, then you need to do the following to get the subsystem definitions:

1. Start the **old** version of WildFly/JBoss AS 7 using `--server-config=standalone-full-ha.xml`
2. Run `org.wildfly.legacy.util.GrabModelVersionsUtil`, which will output the subsystem versions to `target/standalone-model-versions-running.dmr`
3. Run `org.wildfly.legacy.util.DumpStandaloneResourceDefinitionUtil` which will output the full resource definition to `target/standalone-resource-definition-running.dmr`
4. Stop the running version of WildFly/JBoss AS 7

See what changed

To do this follow the following steps

1. Start the **new** version of WildFly using `--server-config=standalone-full-ha.xml`
2. Run `org.wildfly.legacy.util.CompareModelVersionsUtil` and answer the following questions"
    1. Enter Legacy AS version:
        - If it is known version in the `tools/src/test/resources/legacy-models` folder, enter the version number.
        - If it is a not known version, and you got the data yourself in the last step, enter '`running`'
    2. Enter type:
        - Answer 'S'
    3. Read from target directory or from the legacy-models directory:
        - If it is known version in the `controller/src/test/resources/legacy-models` folder, enter '`l`'.
        - If it is a not known version, and you got the data yourself in the last step, enter '`t`'
    4. Report on differences in the model when the management versions are different?:
        - Answer 'y'

Here is some example output, as a subsystem developer you can ignore everything down to `======` `Comparing subsystem models ======`:

```
Enter legacy AS version: 7.2.0.Final
Using target model: 7.2.0.Final
Enter type [S](standalone)/H(host)/D(domain)/F(domain + host):S
Read from target directory or from the legacy-models directory - t/[l]:
Report on differences in the model when the management versions are different? y/[n]: y
Reporting on differences in the model when the management versions are different
Loading legacy model versions for 7.2.0.Final....
Loaded legacy model versions
Loading model versions for currently running server...
Oct 01, 2013 6:26:03 PM org.xnio.Xnio <clinit>
INFO: XNIO version 3.1.0.CR7
Oct 01, 2013 6:26:03 PM org.xnio.nio.NioXnio <clinit>
INFO: XNIO NIO Implementation Version 3.1.0.CR7
Oct 01, 2013 6:26:03 PM org.jboss.remoting3.EndpointImpl <clinit>
INFO: JBoss Remoting version 4.0.0.Beta1
Loaded current model versions
Loading legacy resource descriptions for 7.2.0.Final....
Loaded legacy resource descriptions
Loading resource descriptions for currently running STANDALONE...
Loaded current resource descriptions
Starting comparison of the current....

====== Comparing core models ======
-- SNIP --

====== Comparing subsystem models ======
-- SNIP --
====== Resource root address: ["subsystem" => "remoting"] - Current version: 2.0.0; legacy
version: 1.2.0 =======
--- Problems for relative address to root []:
Missing child types in current: []; missing in legacy [http-connector]
--- Problems for relative address to root ["remote-outbound-connection" => "*"]:
Missing attributes in current: []; missing in legacy [protocol]
Missing parameters for operation 'add' in current: []; missing in legacy [protocol]
-- SNIP --
====== Resource root address: ["subsystem" => "weld"] - Current version: 2.0.0; legacy version:
1.0.0 =======
--- Problems for relative address to root []:
Missing attributes in current: []; missing in legacy [require-bean-descriptor,
non-portable-mode]
Missing parameters for operation 'add' in current: []; missing in legacy
[require-bean-descriptor, non-portable-mode]

Done comparison of STANDALONE!
```

So we can see that for the `remoting` subsystem, we have added a child type called `http-connector`, and we have added an attribute called `protocol` (they are missing in legacy).
in the `weld` subsystem, we have added the `require-bean-descriptor` and `non-portable-mode` attributes in the current version. It will also point out other issues like changed attribute types, changed defaults etc.

Warning

Note that CompareModelVersionsUtil simply inspects the raw resource descriptions of the specified legacy and current models. Its results show the differences between the two. They do not take into account whether one or more transformers have already been written for those versions differences. You will need to check that transformers are not already in place for those versions.

One final point to consider are that some subsystems register runtime-only resources and operations. For example the `modcluster` subsystem has a `stop` method. These do not get registered on the `DC`, e.g. there is no `/profile=full-ha/subsystem=modcluster:stop` operation, it only exists on the servers, for example `/host=xxx/server=server-one/subsystem=modcluster:stop`. What this means is that you don't have to transform such operations and resources. The reason is they are not callable on the DC, and so do not need propagation to the servers in the domain, which in turn means no transformation is needed.

11.10.6 How do I write a transformer?

There are two APIs available to write transformers for a resource. There is the original low-level API where you register transformers directly, the general idea is that you get hold of a `TransformersSubRegistration` for each level and implement the `ResourceTransformer`, `OperationTransformer` and `PathAddressTransformer` interfaces directly. It is, however, a pretty complex thing to do, so we recommend the other approach. For completeness here is the entry point to handling transformation in this way.

```
public class SomeExtension implements Extension {

    private static final String SUBSYSTEM_NAME = "my-subsystem"'

    private static final int MANAGEMENT_API_MAJOR_VERSION = 2;
    private static final int MANAGEMENT_API_MINOR_VERSION = 0;
    private static final int MANAGEMENT_API_MICRO_VERSION = 0;

    @Override
    public void initialize(ExtensionContext context) {
        SubsystemRegistration registration = context.registerSubsystem(SUBSYSTEM_NAME,
MANAGEMENT_API_MAJOR_VERSION,
                MANAGEMENT_API_MINOR_VERSION, MANAGEMENT_API_MICRO_VERSION);
        //Register the resource definitions
        ....
    }

    static void registerTransformers(final SubsystemRegistration subsystem) {
        registerTransformers_1_1_0(subsystem);
        registerTransformers_1_2_0(subsystem);
    }

    /**
     * Registers transformers from the current version to ModelVersion 1.1.0
     */
    private static void registerTransformers_1_1_0(final SubsystemRegistration subsystem) {
        final ModelVersion version = ModelVersion.create(1, 1, 0);

        //The default resource transformer forwards all operations
        final TransformersSubRegistration registration =
subsystem.registerModelTransformers(version, ResourceTransformer.DEFAULT);
        final TransformersSubRegistration child =
registration.registerSubResource(PathElement.pathElement("child"));
        //We can do more things on the TransformersSubRegistation instances


        registerRelayTransformers(stack);
    }
```

Having implemented a number of transformers using the above approach, we decided to simplify things, so we introduced the `org.jboss.as.controller.transform.description.ResourceTransformationDescriptionB` API. It is a lot simpler and avoids a lot of the duplication of functionality required by the low-level API approach. While it doesn't give you the full power that the low-level API does, we found that there are very few places in the WildFly codebase where this does not work, so we will focus on the `ResourceTransformationDescriptionBuilder` API here. (If you come across a problem where this does not work, get in touch with someone from the WildFly Domain Management Team and we should be able to help). The builder API makes all the nasty calls to `TransformersSubRegistration` for you under the hood. It also allows you to fall back to the low-level API in places, although that will not be covered in the current version of this guide. The entry point for using the builder API here is taken from the WeldExtension (in current WildFly this has ModelVersion 2.0.0).

```
private void registerTransformers(SubsystemRegistration subsystem) {
        ResourceTransformationDescriptionBuilder builder =
TransformationDescriptionBuilder.Factory.createSubsystemInstance();
        //These new attributes are assumed to be 'true' in the old version but default to false
in the current version. So discard if 'true' and reject if 'undefined'.
        builder.getAttributeBuilder()
                .setDiscard(new DiscardAttributeChecker.DiscardAttributeValueChecker(false,
false, new ModelNode(true)),
                        WeldResourceDefinition.NON_PORTABLE_MODE_ATTRIBUTE,
WeldResourceDefinition.REQUIRE_BEAN_DESCRIPTOR_ATTRIBUTE)
                .addRejectCheck(new RejectAttributeChecker.DefaultRejectAttributeChecker() {

                    @Override
                    public String getRejectionLogMessage(Map<String, ModelNode> attributes) {
                        return
WeldMessages.MESSAGES.rejectAttributesMustBeTrue(attributes.keySet());
                    }

                    @Override
                    protected boolean rejectAttribute(PathAddress address, String attributeName,
ModelNode attributeValue,
                            TransformationContext context) {
                        //This will not get called if it was discarded, so reject if it is
undefined (default==false) or if defined and != 'true'
                        return !attributeValue.isDefined() ||
!attributeValue.asString().equals("true");
                    }
                }, WeldResourceDefinition.NON_PORTABLE_MODE_ATTRIBUTE,
WeldResourceDefinition.REQUIRE_BEAN_DESCRIPTOR_ATTRIBUTE)
                .end();
        TransformationDescription.Tools.register(builder.build(), subsystem,
ModelVersion.create(1, 0, 0));
    }
```

Here we register a `discard check` and a `reject check`. As mentioned in Attribute transformation lifecycle all attributes are inspected for whether they should be discarded first. Then all attributes which were not discarded are checked for if they should be rejected. We will dig more into what this code means in the next few sections, but in short it means that we discard the `require-bean-descriptor` and `non-portable` attributes on the `weld` subsystem resource if they have the value `true`. If they have any other value, they will not get discarded and so reach the reject check, which will reject the transformation of the attributes if they have any other value.

Here we are saying that we should discard the `require-bean-descriptor` and `non-portable-mode` attributes on the `weld` subsystem resource if they are undefined, and reject them if they are defined. So that means that if the weld subsystem looks like

```
{
        "non-portable-mode" => false,
        "require-bean-descriptor" => false
    }
```

or

```
{
        "non-portable-mode" => undefined,
        "require-bean-descriptor" => undefined
    }
```

or any other combination (the default values for these attributes if undefined is `false`) we will reject the transformation for the slave legacy HC.

If the resource has true for these attributes:

```
{
        "non-portable-mode" => true,
        "require-bean-descriptor" => true
    }
```

they both get discarded (i.e. removed), so they will not get inspected for rejection, and an empty model not containing these attributes gets sent to the legacy HC.

Here we will discuss this API a bit more, to outline the most important features/most commonly needed tasks.

ResourceTransformationDescriptionBuilder

The `ResourceTransformationDescriptionBuilder` contains transformations for a resource type. The initial one is for the subsystem, obtained by the following call:

```
ResourceTransformationDescriptionBuilder subsystemBuilder =
TransformationDescriptionBuilder.Factory.createSubsystemInstance();
```

The `ResourceTransformationDescriptionBuilder` contains functionality for how to handle child resources, which we will look at in this section. It is also the entry point to how to handle transformation of attributes as we will see in AttributeTransformationDescriptionBuilder. Also, it allows you to further override operation transformation as discussed in OperationTransformationOverrideBuilder. When we have finished with our builder, we register it with the `SubsystemRegistration` against the target ModelVersion.

```
TransformationDescription.Tools.register(subsystemBuilder.build(), subsystem,
ModelVersion.create(1, 0, 0));
```

Important

If you have several old ModelVersions you could be transforming to, you need a separate builder for each of those.

Silently discard child resources

To make the `ResourceTransformationDescriptionBuilder` do something, we need to call some of its methods. For example, if we want to silently discard a child resource, we can do

```
subsystemBuilder.discardChildResource(PathElement.pathElement("child", "discarded"));
```

This means that any usage of `/subsystem=my-subsystem/child=discarded` never make it to the legacy slave HC running ModelVersion 1.0.0. During the initial domain model transfer, that part of the serialized domain model is stripped out, and any operations on this address are not forwarded on to the legacy slave HCs running that version of the subsystem. (For brevity this section will leave out the leading `/profile=xxx` part used in domain mode, and use `/subsystem=my-subsystem` as the 'top-level' address).

Warning

Note that discarding, although the simplest option in theory, is **rarely the right thing to do**.

The presence of the defined child normally implies some behaviour on the DC, and that behaviour is not available on the legacy slave HC, so normally rejection is a better policy for those cases. Remember we can have different profiles targeting different groups of versions of legacy slave HCs.

Reject child resource

If we want to reject transformation if a child resource exists, we can do

```
subsystemBuilder.rejectChildResource(PathElement.pathElement("child", "reject"));
```

Now, if there are any legacy slaves running ModelVersion 1.0.0, any usage of `/subsystem=my-subsystem/child=reject` will get rejected for those slaves. Both during the initial domain model transfer, and if any operations are invoked on that address. For example the `remoting` subsystem did not have a `http-connector=*` child until ModelVersion 2.0.0, so it is set up to reject that child when transforming to legacy HCs for all previous ModelVersions (1.1.0, 1.2.0 and 1.3.0). (See Rejection in resource transformers and Rejection in operation transformers for exactly what happens when something is rejected).

Redirect address for child resource

Sometimes we rename the addresses for a child resource between model versions. To do that we use one of the `addChildRedirection()` methods, note that these also return a builder for the child resource (since we are not rejecting or discarding it), we can do this for all children of a given type:

```
ResourceTransformationDescriptionBuilder childBuilder =
       subsystemBuilder.addChildRedirection(PathElement.pathElement("newChild"),
PathElement.pathElement("oldChild");
```

Now, in the initial domain transfer `/subsystem=my-subsystem/newChild=test` becomes `/subsystem=my-subsystem/oldChild=test`. Similarly all operations against the former address get mapped to the latter when executing operations on the DC before sending them to the legacy slave HC running ModelVersion 1.1.0 of the subsystem.

We can also rename a specific named child:

```
ResourceTransformationDescriptionBuilder childBuilder =
       subsystemBuilder.addChildRedirection(PathElement.pathElement("newChild", "newName"),
PathElement.pathElement("oldChild", "oldName");
```

Now, `/subsystem=my-subsystem/newChild=newName` becomes `/subsystem=my-subsystem/oldChild=oldName` both in the initial domain transfer, and when mapping operations to the legacy slave. For example, under the web subsystem `ssl=configuration` got renamed to `configuration=ssl` in later versions, meaning we need a redirect from `configuration=ssl` to `ssl=configuration` in its transformers.

Getting a child resource builder

Sometimes we don't want to transform the subsystem resource, but we want to transform something in one of its child resources. Again, since we are not discarding or rejecting, we get a reference to the builder for the child resource.

```
ResourceTransformationDescriptionBuilder childBuilder =
       subsystemBuilder.addChildResource(PathElement.pathElement("some-child"));
    //We don't actually want to transform anything in /subsystem-my-subsystem/some-child=*
either :-)
    //We are interested in /subsystem-my-subsystem/some-child=*/another-level
    ResourceTransformationDescriptionBuilder anotherBuilder =
       childBuilder.addChildResource(PathElement.pathElement("another-level"));

    //Use anotherBuilder to add child-resource and/or attribute transformation
    ....
```

AttributeTransformationDescriptionBuilder

To transform attributes you call `ResourceTransformationDescriptionBuilder.getAttributeBuilder()` which returns you a `AttributeTransformationDescriptionBuilder` which is used to define transformation for the resource's attributes. For example this gets the attribute builder for the subsystem resource:

```
AttributeTransformationDescriptionBuilder attributeBuilder =
subSystemBuilder.getAttributeBuilder();
```

or we could get it for one of the child resources:

```
ResourceTransformationDescriptionBuilder childBuilder =
       subsystemBuilder.addChildResource(PathElement.pathElement("some-child"));
    AttributeTransformationDescriptionBuilder attributeBuilder =
childBuilder.getAttributeBuilder();
```

The attribute transformations defined by the `AttributeTransformationDescriptionBuilder` will also impact the parameters to all operations defined on the resource. This means that if you have defined the `example` attribute of `/subsystem=my-subsystem/some-child=*` to reject transformation if its value is `true`, the inital domain transfer will reject if it is `true`, also the transformation of the following operations will reject:

```
/subsystem=my-subsystem/some-child=test:add(example=true)
    /subsystem=my-subsystem:write-attribute(name=example, value=true)
    /subsystem=my-subsystem:custom-operation(example=true)
```

The following operations will pass in this example, since the `example` attribute is not getting set to `true`

```
/subsystem=my-subsystem/some-child=test:add(example=false)
    /subsystem=my-subsystem/some-child=test:add()             //Here it 'example' is simply left
undefined
    /subsystem=my-subsystem:write-attribute(name=example, value=false)
    /subsystem=my-subsystem:undefine-attribute(name=example)  //Again this makes 'example'
undefined
    /subsystem=my-subsystem:custom-operation(example=false)
```

For the rest of the examples in this section we assume that the `attributeBuilder` is for `/subsystem=my-subsystem`

Attribute transformation lifecycle

There is a well defined lifecycle used for attribute transformation that is worth explaining before jumping into specifics. Transformation is done in the following phases, in the following order:

1. `discard` - All attributes in the domain model transfer or invoked operation that have been registered for a discard check, are checked to see if the attribute should be discarded. If an attribute should be discarded, it is removed from the resource's attributes/operation's parameters and it does not get passed to the next phases. Once discarded it does not get sent to the legacy slave HC.
2. `reject` - All attributes that have been registered for a reject check (and which not have been discarded) are checked to see if the attribute should be rejected. As explained in Rejection in resource transformers and Rejection in operation transformers exactly what happens when something is rejected varies depending on whether we are transforming a resource or an operation, and the version of the legacy slave HC we are transforming for. If a transformer rejects an attribute, all other reject transformers still get invoked, and the next phases also get invoked. This is because we don't know in all cases what will happen if a reject happens. Although this might sound cumbersome, in practice it actually makes it easier to write transformers since you only need one kind regardless of if it is a resource, an operation, and legacy slave HC version. However, as we will see in Common transformation use-cases, it means some extra checks are needed when writing reject and convert transformers.
3. `convert` - All attributes that have been registered for conversion are checked to see if the attribute should be converted. If the attribute does not exist in the original operation/resource it may be introduced. This is useful for setting default values for the target legacy slave HC.
4. `rename` - All attributes registered for renaming are renamed.

Next, let us have a look at how to register attributes for each of these phases.

Discarding attributes

The general idea behind a discard is that we remove attributes which do not exist in the legacy slave HC's model. However, as hopefully described below, we normally can't simply discard everything, we need to check the values first.

To discard an attribute we need an instance of `org.jboss.as.controller.transform.description.DiscardAttributeChecker`, and call the following method on the `AttributeTransformationDescriptionBuilder`:

```
DiscardAttributeChecker discardCheckerA = ....;
     attributeBuilder.setDiscard(discardCheckerA, "attr1", "attr2");
```

As shown, you can register the `DiscardAttributeChecker` for several attributes at once, in the above example both `attr1` and `attr2` get checked for if they should be discarded. You can also register different `DiscardAttributeChecker` instances for different attributes:

```
DiscardAttributeChecker discardCheckerA = ....;
     DiscardAttributeChecker discardCheckerB = ....;
     attributeBuilder.setDiscard(discardCheckerA, "attr1");
     attributeBuilder.setDiscard(discardCheckerA, "attr2");
```

Note that you can only have one `DiscardAttributeChecker` per attribute, so the following would cause an error (if running with assertions enabled, otherwise `discardCheckerB` will overwrite `discardCheckerA`):

```
DiscardAttributeChecker discardCheckerA = ....;
     DiscardAttributeChecker discardCheckerB = ....;
     attributeBuilder.setDiscard(discardCheckerA, "attr1");
     attributeBuilder.setDiscard(discardCheckerB, "attr1");
```

The DiscardAttributeChecker interface

`org.jboss.as.controller.transform.description.DiscardAttributeChecker` contains both the `DiscardAttributeChecker` and some helper implementations. The implementations of this interface get called for each attribute they are registered against. The interface itself is quite simple:

```
public interface DiscardAttributeChecker {

    /**
     * Returns {@code true} if the attribute should be discarded if expressions are used
     *
     * @return whether to discard if expressions are used
     */
    boolean isDiscardExpressions();
```

Return `true` here to discard the attribute if it is an expression. If it is an expression, and this method returns `true`, the `isOperationParameterDiscardable` and `isResourceAttributeDiscardable` methods will not get called.

```
/**
     * Returns {@code true} if the attribute should be discarded if it is undefined
     *
     * @return whether to discard if the attribute is undefined
     */
    boolean isDiscardUndefined();
```

Return `true` here to discard the attribute if it is `undefined`. If it is `undefined`, and this method returns `true`, the `isDiscardExpressions`, `isOperationParameterDiscardable` and `isResourceAttributeDiscardable` methods will not get called.

```
/**
     * Gets whether the given operation parameter can be discarded
     *
     * @param address the address of the operation
     * @param attributeName the name of the operation parameter.
     * @param attributeValue the value of the operation parameter.
     * @param operation the operation executed. This is unmodifiable.
     * @param context the context of the transformation
     *
     * @return {@code true} if the operation parameter value should be discarded, {@code false}
otherwise.
     */
    boolean isOperationParameterDiscardable(PathAddress address, String attributeName, ModelNode
attributeValue, ModelNode operation, TransformationContext context);
```

If we are transforming an operation, this method gets called for each operation parameter. We have access to the address of the operation, the name and value of the operation parameter, an unmodifiable copy of the original operation and the `TransformationContext`. The `TransformationContext` allows you access to the original resource the operation is working on before any transformation happened, which is useful if you want to check other values in the resource if this is, say a `write-attribute` operation. Return `true` to discard the operation.

```
/**
     * Gets whether the given attribute can be discarded
     *
     * @param address the address of the resource
     * @param attributeName the name of the attribute
     * @param attributeValue the value of the attribute
     * @param context the context of the transformation
     *
     * @return {@code true} if the attribute value should be discarded, {@code false} otherwise.
     */
    boolean isResourceAttributeDiscardable(PathAddress address, String attributeName, ModelNode
attributeValue, TransformationContext context);
```

If we are transforming a resource, this method gets called for each attribute in the resource. We have access to the address of the resource, the name and value of the attribute, and the `TransformationContext`. Return `true` to discard the operation.

```
}
```

DiscardAttributeChecker helper classes/implementations

`DiscardAttributeChecker` contains a few helper implementations for the most common cases to save you writing the same stuff again and again.

DiscardAttributeChecker.DefaultDiscardAttributeChecker

`DiscardAttributeChecker.DefaultDiscardAttributeChecker` is an abstract convenience class. In most cases you don't need a separate check for if an operation or a resource is being transformed, so it makes both the `isResourceAttributeDiscardable()` and `isOperationParameterDiscardable()` methods call the following method.

```
protected abstract boolean isValueDiscardable(PathAddress address, String attributeName,
ModelNode attributeValue, TransformationContext context);
```

All you lose, in the case of an operation transformation, is the name of the transformed operation. The constructor of `DiscardAttributeChecker.DefaultDiscardAttributeChecker` also allows you to define values for `isDiscardExpressions()` and `isDiscardUndefined()`.

DiscardAttributeChecker.DiscardAttributeValueChecker

This is another convenience class, which allows you to discard an attribute if it has one or more values. Here is a real-world example from the `jpa` subsystem:

```
private void initializeTransformers_1_1_0(SubsystemRegistration subsystemRegistration) {
        ResourceTransformationDescriptionBuilder builder =
TransformationDescriptionBuilder.Factory.createSubsystemInstance();
        builder.getAttributeBuilder()
            .setDiscard(
                  new DiscardAttributeChecker.DiscardAttributeValueChecker(new
ModelNode(ExtendedPersistenceInheritance.DEEP.toString())),
                  JPADefinition.DEFAULT_EXTENDEDPERSISTENCE_INHERITANCE)
            .addRejectCheck(RejectAttributeChecker.DEFINED,
JPADefinition.DEFAULT_EXTENDEDPERSISTENCE_INHERITANCE)
            .end();
        TransformationDescription.Tools.register(builder.build(), subsystemRegistration,
ModelVersion.create(1, 1, 0));
    }
```

We will come back to the reject checks in the Rejecting attributes section. We are saying that we should discard the `JPADefinition.DEFAULT_EXTENDEDPERSISTENCE_INHERITANCE` attribute if it has the value `deep`. The reasoning here is that this attribute did not exist in the old model, but the legacy slave HCs *implied behaviour* is that this was `deep`. In the current version we added the possibility to toggle this setting, but only `deep` is consistent with what is available in the legacy slave HC. In this case we are using the constructor for `DiscardAttributeChecker.DiscardAttributeValueChecker` which says don't discard if it uses expressions, and discard if it is `undefined`. If it is `undefined` in the current model, looking at the default value of `JPADefinition.DEFAULT_EXTENDEDPERSISTENCE_INHERITANCE`, it is `deep`, so a discard is in line with the implied legacy behaviour. If an expression is used, we cannot discard since we have no idea what the expression will resolve to on the slave HC.

DiscardAttributeChecker.ALWAYS

`DiscardAttributeChecker.ALWAYS` will always discard an attribute. Use this sparingly, since normally the presence of an attribute in the current model implies some behaviour should be turned on, and if that does not exist in the legacy model it implies that that behaviour does not exist in the legacy slave HC and its servers. Normally the legacy slave HC's subsystem has some implied behaviour which is better checked for by using a `DiscardAttributeChecker.DiscardAttributeValueChecker`. One valid use for `DiscardAttributeChecker.ALWAYS` can be found in the `ejb3` subsystem:

```
private static void registerTransformers_1_1_0(SubsystemRegistration subsystemRegistration) {
        ResourceTransformationDescriptionBuilder builder =
TransformationDescriptionBuilder.Factory.createSubsystemInstance()
                .getAttributeBuilder()
                 ...
                // We can always discard this attribute, because it's meaningless without the
security-manager subsystem, and
                // a legacy slave can't have that subsystem in its profile.
                .setDiscard(DiscardAttributeChecker.ALWAYS,
EJB3SubsystemRootResourceDefinition.DISABLE_DEFAULT_EJB_PERMISSIONS)
   ...
```

As the comment says, this attribute only makes sense with the security-manager susbsystem, which does not exist on legacy slaves running ModelVersion 1.1.0 of the `ejb3` subsystem.

DiscardAttributeChecker.UNDEFINED

`DiscardAttributeChecker.UNDEFINED` will discard an attribute if it is `undefined`. This is normally safer than `DiscardAttributeChecker.ALWAYS` since the attribute is not set in the current model, we don't need to send it to the legacy model. However, you should check that this attribute not existing in the legacy slave HC, implies the same functionality as being undefined in the current DC.

Rejecting attributes

The next step is to check attributes and values which we know for sure will not work on the target legacy slave HC.

To reject an attribute we need an instance of `org.jboss.as.controller.transform.description.RejectAttributeChecker`, and call the following method on the `AttributeTransformationDescriptionBuilder`:

```
RejectAttributeChecker rejectCheckerA = ....;
     attributeBuilder.addRejectCheck(rejectCheckerA, "attr1", "attr2");
```

As shown you can register the `RejectAttributeChecker` for several attributes at once, in the above example both `attr1` and `attr2` get checked for if they should be discarded. You can also register different `RejectAttributeChecker` instances for different attributes:

```
RejectAttributeChecker rejectCheckerA = ....;
     RejectAttributeChecker rejectCheckerB = ....;
     attributeBuilder.addRejectCheck(rejectCheckerA, "attr1");
     attributeBuilder.addRejectCheck(rejectCheckerB, "attr2");
```

You can also register several `RejectAttributeChecker` instances per attribute

```
RejectAttributeChecker rejectCheckerA = ....;
     RejectAttributeChecker rejectCheckerB = ....;
     attributeBuilder.addRejectCheck(rejectCheckerA, "attr1");
     attributeBuilder.addRejectCheck(rejectCheckerB, "attr1, "attr2");
```

In this case `attr1` gets both `rejectCheckerA` and `rejectCheckerB`. For attributes with several `RejectAttributeChecker` registered, they get processed in the order that they have been added. So when checking `attr1` for rejection, `rejectCheckerA` gets run before `rejectCheckerB`. As mentioned in Attribute transformation lifecycle, if an attribute is rejected, we still invoke the rest of the reject checkers.

The RejectAttributeChecker interface

`org.jboss.as.controller.transform.description.RejectAttributeChecker` contains both the `RejectAttributeChecker` and some helper implementations. The implementations of this interface get called for each attribute they are registered against. The interface itself is quite simple, and its main methods are similar to `DiscardAttributeChecker`:

```
public interface RejectAttributeChecker {
    /**
     * Determines whether the given operation parameter value is not understandable by the
target process and needs
     * to be rejected.
     *
     * @param address        the address of the operation
     * @param attributeName  the name of the attribute
     * @param attributeValue the value of the attribute
     * @param operation      the operation executed. This is unmodifiable.
     * @param context        the context of the transformation
     * @return {@code true} if the parameter value is not understandable by the target process
and so needs to be rejected, {@code false} otherwise.
     */
    boolean rejectOperationParameter(PathAddress address, String attributeName, ModelNode
attributeValue, ModelNode operation, TransformationContext context);
```

If we are transforming an operation, this method gets called for each operation parameter. We have access to the address of the operation, the name and value of the operation parameter, an unmodifiable copy of the original operation and the `TransformationContext`. The `TransformationContext` allows you access to the original resource the operation is working on before any transformation happened, which is useful if you want to check other values in the resource if this is, say a `write-attribute` operation. Return `true` to reject the operation.

```
/**
     * Gets whether the given resource attribute value is not understandable by the target
process and needs
     * to be rejected.
     *
     * @param address        the address of the resource
     * @param attributeName  the name of the attribute
     * @param attributeValue the value of the attribute
     * @param context        the context of the transformation
     * @return {@code true} if the attribute value is not understandable by the target process
and so needs to be rejected, {@code false} otherwise.
     */
    boolean rejectResourceAttribute(PathAddress address, String attributeName, ModelNode
attributeValue, TransformationContext context);
```

If we are transforming a resource, this method gets called for each attribute in the resource. We have access to the address of the resource, the name and value of the attribute, and the `TransformationContext`. Return `true` to discard the operation.

```
/**
     * Returns the log message id used by this checker. This is used to group it so that all
attributes failing a type of rejection
     * end up in the same error message
     *
     * @return the log message id
     */
    String getRejectionLogMessageId();
```

Here we need a unique id for the log message from the `RejectAttributeChecker`. It is used to group rejected attributes by their log message. A typical implementation will contain {{return getRejectionLogMessage(Collections.<String, ModelNode>emptyMap());}

```
/**
     * Gets the log message if the attribute failed rejection
     *
     * @param attributes a map of all attributes failed in this checker and their values
     * @return the formatted log message
     */
    String getRejectionLogMessage(Map<String, ModelNode> attributes);
```

Here we return a message saying why the attributes were rejected, with the possibility to format the message to include the names of all the rejected attributes and the values they had.

```
}
```

RejectAttributeChecker helper classes/implementations

`RejectAttributeChecker` contains a few helper classes for the most common scenarios to save you from writing the same stuff again and again.

RejectAttributeChecker.DefaultRejectAttributeChecker

`RejectAttributeChecker.DefaultRejectAttributeChecker` is an abstract convenience class. In most cases you don't need a separate check for if an operation or a resource is being transformed, so it makes both the `rejectOperationParameter()` and `rejectResourceAttribute()` methods call the following method.

```
protected abstract boolean rejectAttribute(PathAddress address, String attributeName, ModelNode
attributeValue, TransformationContext context);
```

Like `DefaultDiscardAttributeChecker`, all you loose is the name of the transformed operation, in the case of operation transformation.

RejectAttributeChecker.DEFINED

`RejectAttributeChecker.DEFINED` is used to reject any attribute that has a defined value. Normally this is because the attribute does not exist on the target legacy slave HC. A typical use case for these is for the *implied behavior* example we looked at in the `jpa` subsystem in DiscardAttributeChecker.DiscardAttributeValueChecker

```
private void initializeTransformers_1_1_0(SubsystemRegistration subsystemRegistration) {
        ResourceTransformationDescriptionBuilder builder =
TransformationDescriptionBuilder.Factory.createSubsystemInstance();
        builder.getAttributeBuilder()
            .setDiscard(
                  new DiscardAttributeChecker.DiscardAttributeValueChecker(new
ModelNode(ExtendedPersistenceInheritance.DEEP.toString())),
                  JPADefinition.DEFAULT_EXTENDEDPERSISTENCE_INHERITANCE)
            .addRejectCheck(RejectAttributeChecker.DEFINED,
JPADefinition.DEFAULT_EXTENDEDPERSISTENCE_INHERITANCE)
            .end();
        TransformationDescription.Tools.register(builder.build(), subsystemRegistration,
ModelVersion.create(1, 1, 0));
    }
```

So we discard the `JPADefinition.DEFAULT_EXTENDEDPERSISTENCE_INHERITANCE` value if it is not an expression, and also has the value `deep`. Now if it was not discarded, it would will still be defined so we reject it.

Important

Reject and discard often work in pairs.

RejectAttributeChecker.SIMPLE_EXPRESSIONS

`RejectAttributeChecker.SIMPLE_EXPRESSIONS` can be used to reject an attribute that contains expressions. This was used a lot for transformations to subsystems in JBoss AS 7.1.x, since we had not fully realized the importance of where to support expressions until JBoss AS 7.2.0 was released, so a lot of attributes in earlier versions were missing expressions support.

RejectAttributeChecker.ListRejectAttributeChecker

The `RejectAttributeChecker}}s we have seen so far work on simple attributes, i.e. where the attribute has a ModelType which is one of the primitives. We also have a {{RejectAttributeChecker.ListRejectAttributeChecker` which allows you to define a checker for the elements of a list, when the type of an attribute is `ModelType.LIST`.

```
attributeBuilder
            .addRejectCheck(new ListRejectAttributeChecker(RejectAttributeChecker.EXPRESSIONS),
"attr1");
```

For `attr1` it will check each element of the list and run `RejectAttributeChecker.EXPRESSIONS` to check that each element is not an expression. You can of course pass in another kind of `RejectAttributeChecker` to check the elements as well.

RejectAttributeChecker.ObjectFieldsRejectAttributeChecker

For attributes where the type is `ModelType.OBJECT` we have `RejectAttributeChecker.ObjectFieldsRejectAttributeChecker` which allows you to register different reject checkers for the different fields of the registered object.

```
Map<String, RejectAttributeChecker> fieldRejectCheckers = new HashMap<String,
RejectAttributeChecker>();
    fieldRejectCheckers.put("time", RejectAttributeChecker.SIMPLE_EXPRESSIONS);
    fieldRejectCheckers.put("unit", "Lunar Month");
    attributeBuilder
            .addRejectCheck(new ObjectFieldsRejectAttributeChecker(fieldRejectCheckers),
"attr1");
```

Now if `attr1` is a complex type where `attr1.get("time").getType() == ModelType.EXPRESSION` or `attr1.get("unit").asString().equals("Lunar Month")` we reject the attribute.

Converting attributes

To convert an attribute you register an `org.jboss.as.controller.transform.description.AttributeConverter` instance against the attributes you want to convert:

```
AttributeConverter converterA = ...;
    AttributeConverter converterB = ...;
    attributeBuilder
            .setValueConverter(converterA, "attr1", "attr2");
    attributeBuilder
            .setValueConverter(converterB, "attr3");
```

Now if `attr1` and `attr2` get converted with `converterA`, while `attr3` gets converted with `converterB`.

The AttributeConverter interface

The `AttributeConverter` interface gets called for each attribute for which the `AttributeConverter` has been registered

```
public interface AttributeConverter {

    /**
     * Converts an operation parameter
     *
     * @param address the address of the operation
     * @param attributeName the name of the operation parameter
     * @param attributeValue the value of the operation parameter to be converted
     * @param operation the operation executed. This is unmodifiable.
     * @param context the context of the transformation
     */
    void convertOperationParameter(PathAddress address, String attributeName, ModelNode
attributeValue, ModelNode operation, TransformationContext context);
```

If we are transforming an operation, this method gets called for each operation parameter for which the con. We have access to the address of the operation, the name and value of the operation parameter, an unmodifiable copy of the original operation and the `TransformationContext`. The `TransformationContext` allows you access to the original resource the operation is working on before any transformation happened, which is useful if you want to check other values in the resource if this is, say a write-attribute operation. To change the attribute value, you modify the `attributeValue`.

```
/**
     * Converts a resource attribute
     *
     * @param address the address of the operation
     * @param attributeName the name of the attribute
     * @param attributeValue the value of the attribute to be converted
     * @param context the context of the transformation
     */
    void convertResourceAttribute(PathAddress address, String attributeName, ModelNode
attributeValue, TransformationContext context);
```

If we are transforming a resource, this method gets called for each attribute in the resource. We have access to the address of the resource, the name and value of the attribute, and the `TransformationContext`. To change the attribute value, you modify the `attributeValue`.

```
}
```

A hypothetical example is if the current and legacy subsystems both contain an attribute called `timeout`. In the legacy model this was specified to be milliseconds, however in the current model it has been changed to be seconds, hence we need to convert the value when sending it to slave HCs using the legacy model:

```
AttributeConverter secondsToMs = new AttributeConverter.DefaultAttributeConverter() {
                  @Override
                  protected void convertAttribute(PathAddress address, String attributeName,
ModelNode attributeValue,
                            TransformationContext context) {
                       if (attributeValue.isDefined()) {
                            int seconds = attributeValue.asInt();
                            int milliseconds = seconds * 1000;
                            attributeValue.set(milliseconds);
                       }
                  }
          };

     attributeBuilder.
          .setValueConverter(secondsToMs , "timeout")
```

We need to be a bit careful here. If the `timeout` attribute is an expression our nice conversion will not work, so we need to add a reject check to make sure it is not an expression as well:

```
attributeBuilder.
          .addRejectCheck(SIMPLE_EXPRESSIONS, "timeout")
          .setValueConverter(secondsToMs , "timeout")
```

Now it should be fine.

`AttributeConverter.DefaultAttributeConverter` is is an abstract convenience class. In most cases you don't need a separate check for if an operation or a resource is being transformed, so it makes both the convertOperationParameter() and convertResourceAttribute() methods call the following method.

```
protected abstract void convertAttribute(PathAddress address, String attributeName, ModelNode
attributeValue, TransformationContext context);
```

Like `DefaultDiscardAttributeChecker` and `DefaultRejectAttributeChecker`, all you loose is the name of the transformed operation, in the case of operation transformation.

Introducing attributes during transformation

Say both the current and the legacy models have an attribute called `port`. In the legacy version this attribute had to be specified, and the default xml configuration had `1234` for its value. In the current version this attribute has been made optional with a default value of `1234` so that it does not need to be specified. When transforming to a slave HC using the old version we will need to introduce this attribute if the new model does not contain it:

```
attributeBuilder.
         setValueConverter(AttributeConverter.Factory.createHardCoded(new ModelNode(1234) true),
"port");
```

So what this factory method does is to create an implementation of `AttributeConverter.DefaultAttributeConverter` where in `convertAttribute()` we set `attributeValue` to have the value `1234` if it is `undefined`. As long as `attributeValue` gets set in that method it will get set in the model, regardless of if it existed already or not.

Renaming attributes

To rename an attribute, you simply do

```
attributeBuilder.addRename("my-name", "legacy-name");
```

Now, in the initial domain transfer to the legacy slave HC, we rename `/subsystem=my-subsystem`'s `my-name` attribute to `legacy-name`. Also, the operations involving this attribute are affected, so

```
/subsystem=my-subsystem/:add(my-name=true)  ->
         /subsystem=my-subsystem/:add(legacy-name=true)
    /subsystem=my-subsystem:write-attribute(name=my-name, value=true) ->
         /subsystem=my-subsystem:write-attribute(name=legacy-name, value=true)
    /subsystem=my-subsystem:undefine-attribute(name=my-name) ->
         /subsystem=my-subsystem:undefine-attribute(name=legacy-name)
```

OperationTransformationOverrideBuilder

All operations on a resource automatically get the same transformations on their parameters as set up by the `AttributeTransformationDescriptionBuilder`. In some cases you might want to change this, so you can use the `OperationTransformationOverrideBuilder`, which is got from:

```
OperationTransformationOverrideBuilder operationBuilder =
subSystemBuilder.addOperationTransformationOverride("some-operation");
```

In this case the operation will now no longer inherit the attribute/operation parameter transformations, so they are effectively turned off. In other cases you might want to include them by calling `inheritResourceAttributeDefinitions()`, and to include some more checks (the `OperationTransformationBuilder` interface has all the methods found in `AttributeTransformationBuilder`:

```
OperationTransformationOverrideBuilder operationBuilder =
subSystemBuilder.addOperationTransformationOverride("some-operation");
    operationBuilder.inheritResourceAttributeDefinitions();
    operationBuilder.setValueConverter(AttributeConverter.Factory.createHardCoded(new
ModelNode(1234) true), "port");
```

You can also rename operations, in this case the operation `some-operation` gets renamed to `legacy-operation` before getting sent to the legacy slave HC.

```
OperationTransformationOverrideBuilder operationBuilder =
subSystemBuilder.addOperationTransformationOverride("some-operation");
    operationBuilder.rename("legacy-operation");
```

11.10.7 Evolving transformers with subsystem ModelVersions

Say you have a subsystem with ModelVersions 1.0.0 and 1.1.0. There will (hopefully!) already be transformers in place for 1.1.0 to 1.0.0 transformations. Let's say that the transformers registration looks like:

```
public class SomeExtension implements Extension {

    private static final String SUBSYSTEM_NAME = "my-subsystem"'

    private static final int MANAGEMENT_API_MAJOR_VERSION = 1;
    private static final int MANAGEMENT_API_MINOR_VERSION = 1;
    private static final int MANAGEMENT_API_MICRO_VERSION = 0;

    @Override
    public void initialize(ExtensionContext context) {
        SubsystemRegistration registration = context.registerSubsystem(SUBSYSTEM_NAME,
MANAGEMENT_API_MAJOR_VERSION,
                MANAGEMENT_API_MINOR_VERSION, MANAGEMENT_API_MICRO_VERSION);
        //Register the resource definitions
        ....
    }

    private void registerTransformers(final SubsystemRegistration subsystem) {
        registerTransformers_1_0_0(subsystem);
    }

    /**
     * Registers transformers from the current version to ModelVersion 1.0.0
     */
    private void registerTransformers_1_0_0(SubsystemRegistration subsystem) {
        ResourceTransformationDescriptionBuilder builder =
TransformationDescriptionBuilder.Factory.createSubsystemInstance();
        builder.getAttributeBuilder()
            .addRejectCheck(RejectAttributeChecker.DEFINED, "attr1")
            .end();
        TransformationDescription.Tools.register(builder.build(), subsystem,
ModelVersion.create(1, 0, 0));
    }
}
```

Now say we want to do a new version of the model. This new version contains a new attribute called 'new-attr' which cannot be defined when transforming to 1.1.0, we bump the model version to 2.0.0:

```
public class SomeExtension implements Extension {

    private static final String SUBSYSTEM_NAME = "my-subsystem"'

    private static final int MANAGEMENT_API_MAJOR_VERSION = 2;
    private static final int MANAGEMENT_API_MINOR_VERSION = 0;
    private static final int MANAGEMENT_API_MICRO_VERSION = 0;

    @Override
    public void initialize(ExtensionContext context) {
        SubsystemRegistration registration = context.registerSubsystem(SUBSYSTEM_NAME,
MANAGEMENT_API_MAJOR_VERSION,
                MANAGEMENT_API_MINOR_VERSION, MANAGEMENT_API_MICRO_VERSION);
        //Register the resource definitions
        ....
    }
```

There are a few ways to evolve your transformers:

- The old way
- Chained transformers

The old way

This is the way that has been used up to WildFly 8.x. However, in WildFly 9 and later, it is strongly recommended to migrate to what is mentioned in Chained transformers

Now we need some new transformers from the current ModelVersion to 1.1.0 where we reject any defined occurrances of our new attribute `new-attr`:

```
private void registerTransformers(final SubsystemRegistration subsystem) {
        registerTransformers_1_0_0(subsystem);
        registerTransformers_1_1_0(subsystem);
    }

    /**
     * Registers transformers from the current version to ModelVersion 1.1.0
     */
    private void registerTransformers_1_1_0(SubsystemRegistration subsystem) {
        ResourceTransformationDescriptionBuilder builder =
TransformationDescriptionBuilder.Factory.createSubsystemInstance();
        builder.getAttributeBuilder()
            .addRejectCheck(RejectAttributeChecker.DEFINED, "new-attr")
            .end();
        TransformationDescription.Tools.register(builder.build(), subsystem,
ModelVersion.create(1, 1, 0));
    }
```

So that is all well and good, however we also need to take into account that `new-attr` **does not exist in ModelVersion 1.0.0 either**, so we need to extend our transformer for 1.0.0 to reject it there as well. As you can see 1.0.0 also rejects a defined 'attr1' in addition to the 'new-attr'(which is rejected in both versions).

```
/**
     * Registers transformers from the current version to ModelVersion 1.0.0
     */
    private void registerTransformers_1_0_0(SubsystemRegistration subsystem) {
        ResourceTransformationDescriptionBuilder builder =
TransformationDescriptionBuilder.Factory.createSubsystemInstance();
        builder.getAttributeBuilder()
            .addRejectCheck(RejectAttributeChecker.DEFINED, "attr1", "new-attr")
            .end();
        TransformationDescription.Tools.register(builder.build(), subsystem,
ModelVersion.create(1, 0, 0));
    }
}
```

Now `new-attr` will be rejected if defined for all previous model versions.

Chained transformers

Since 'The old way' had a lot of duplication of code, since WildFly 9 we now have chained transformers. You obtain a `ChainedTransformationDescriptionBuilder` which is a different entry point to the `ResourceTransformationDescriptionBuilder` we have seen earlier. Each `ResourceTransformationDescriptionBuilder` deals with transformation across one version delta.

```
private void registerTransformers(SubsystemRegistration subsystem) {
        ModelVersion version1_1_0 = ModelVersion.create(1, 1, 0);
        ModelVersion version1_0_0 = ModelVersion.create(1, 0, 0);

        ChainedTransformationDescriptionBuilder chainedBuilder =

TransformationDescriptionBuilder.Factory.createChainedSubystemInstance(subsystem.getSubsystemVersi
//Differences between the current version and 1.1.0
        ResourceTransformationDescriptionBuilder builder110 =
            chainedBuilder.create(subsystem.getSubsystemVersion(), version1_1_0);
        builder110.getAttributeBuilder()
            .addRejectCheck(RejectAttributeChecker.DEFINED, "new-attr")
            .end();

        //Differences between the 1.1.0 and 1.0.0
        ResourceTransformationDescriptionBuilder builder100 =
            chainedBuilder.create(subsystem.getSubsystemVersion(), version1_0_0);
        builder110.getAttributeBuilder()
            .addRejectCheck(RejectAttributeChecker.DEFINED, "attr1")
            .end();

        chainedBuilder.buildAndRegister(subsystem, new ModelVersion[]{version1_0_0,
version1_1_0});
```

The `buildAndRegister(ModelVersion[]... chains)` method registers a chain consisting of the built `builder110` and `builder100` for transformation to 1.0.0, and a chain consisting of the built `builder110` for transformation to 1.1.0. It allows you to specify more than one chain.

Now when transforming from the current version to 1.0.0, the resource is first transformed from the current version to 1.1.0 (which rejects a defined `new-attr`) and then it is transformed from 1.1.0 to 1.0.0 (which rejects a defined `attr1`). So when evolving transformers you should normally only need to add things to the last version delta. The full current-to-1.1.0 transformation is run before the 1.1.0-to-1.0.0 transformation is run.

One thing worth pointing out that the value returned by `TransformationContext.readResource(PathAddress address)` and `TransformationContext.readResourceFromRoot(PathAddress address)` which you can use from your custom `RejectAttributeChecker`, `DiscardAttributeChecker` and `AttributeConverter` behaves slightly differently depending on if you are transforming an operation or a resource.

During *resource transformation* this will be the latest model, so in our above example, in the current-to-1.1.0 transformation it will be the original model. In the 1.1.0-to-1.0.0 transformation, it will be the result of the current-to-1.1.0 transformation.

During *operation transformation* these methods will always return the original model (we are transforming operations, not resources!).

In WildFly 9 we are now less aggressive about transforming to all previous versions of WildFly, however we still have a lot of good tests for running against 7.1.x, 8. Also, for Red Hat employees we have tests against EAP versions. These tests no longer get run by default, to run them you need to specify some system properties when invoking maven. They are:

- `-Djboss.test.transformers.subsystem.old` - enables the non-default subsystem tests.
- -Djboss.test.transformers.eap - (Red Hat developers only), enables the eap tests, but only the ones run by default. If run in conjunction with `-Djboss.test.transformers.subsystem.old` you get all the possible subsystem tests run.
- -Djboss.test.transformers.core.old - enables the non-default core model tests.

11.10.8 Testing transformers

To test transformation you need to extend `org.jboss.as.subsystem.test.AbstractSubsystemTest` or `org.jboss.as.subsystem.test.AbstractSubsystemBaseTest`. Then, in order to have the best test coverage possible, you should test the fullest configuration that will work, and you should also test configurations that don't work if you have rejecting transformers registered. The following example is from the threads subsystem, and I have only included the tests against 7.1.2 - there are more! First we need to set up our test:

```
public class ThreadsSubsystemTestCase extends AbstractSubsystemBaseTest {
    public ThreadsSubsystemTestCase() {
        super(ThreadsExtension.SUBSYSTEM_NAME, new ThreadsExtension());
    }

    @Override
    protected String getSubsystemXml() throws IOException {
        return readResource("threads-subsystem-1_1.xml");
    }
```

So we say that this test is for the `threads` subsystem, and that it is implemented by `ThreadsExtension`. This is the same test framework as we use in Example subsystem#Testing the parsers, but we will only talk about the parts relevant to transformers here.

Testing a configuration that works

To test a configuration xxx

```
@Test
    public void testTransformerAS712() throws Exception {
        testTransformer_1_0(ModelTestControllerVersion.V7_1_2_FINAL);
    }
    /**
     * Tests transformation of model from 1.1.0 version into 1.0.0 version.
     *
     * @throws Exception
     */
    private void testTransformer_1_0(ModelTestControllerVersion controllerVersion) throws
Exception {
        String subsystemXml = "threads-transform-1_0.xml";   //This has no expressions not
understood by 1.0
        ModelVersion modelVersion = ModelVersion.create(1, 0, 0); //The old model version
        //Use the non-runtime version of the extension which will happen on the HC
        KernelServicesBuilder builder =
createKernelServicesBuilder(AdditionalInitialization.MANAGEMENT)
                .setSubsystemXmlResource(subsystemXml);

        final PathAddress subsystemAddress =
PathAddress.pathAddress(PathElement.pathElement(SUBSYSTEM, mainSubsystemName));

        // Add legacy subsystems
        builder.createLegacyKernelServicesBuilder(null, controllerVersion, modelVersion)
                .addOperationValidationResolve("add",
subsystemAddress.append(PathElement.pathElement("thread-factory")))
                .addMavenResourceURL("org.jboss.as:jboss-as-threads:" +
controllerVersion.getMavenGavVersion())
                .excludeFromParent(SingleClassFilter.createFilter(ThreadsLogger.class));

        KernelServices mainServices = builder.build();
        KernelServices legacyServices = mainServices.getLegacyServices(modelVersion);
        Assert.assertNotNull(legacyServices);
        checkSubsystemModelTransformation(mainServices, modelVersion);
    }
```

What this test does is get the builder to configure the test controller using `threads-transform-1_0.xml`. This main builder works with the current subsystem version, and the jars in the WildFly checkout.

Next we configure a 'legacy' controller. This will run the version of the core libraries (e.g the `controller` module) as found in the targeted legacy version of JBoss AS/Wildfly), and the subsystem. We need to pass in that it is using the core AS version 7.1.2.Final (i.e. the `ModelTestControllerVersion.V7_1_2_FINAL` part) and that that version is ModelVersion 1.0.0. Next we have some `addMavenResourceURL()` calls passing in the Maven GAVs of the old version of the subsystem and any dependencies it has needed to boot up. Normally, specifying just the Maven GAV of the old version of the subsystem is enough, but that depends on your subsystem. In this case the old subsystem GAV is enough. When booting up the legacy controller the framework uses the parsed operations from the main controller and transforms them using the 1.0.0 transformer in the threads subsystem. The `addOperationValidationResolve()` and `excludeFromParent()` calls are not normally necessary, see the javadoc for more examples.

The call to `KernelServicesBuilder.build()` will build both the main controller and the legacy controller. As part of that it also boots up a second copy of the main controller using the transformed operations to make sure that the 'old' ops to boot our subsystem will still work on the current controller, which is important for backwards compatibility of CLI scripts. To tweak how that is done if you see failures there, see `LegacyKernelServicesInitializer.skipReverseControllerCheck()` and `LegacyKernelServicesInitializer.configureReverseControllerCheck()`. The `LegacyKernelServicesInitializer` is what gets returned by `KernelServicesBuilder.createLegacyKernelServicesBuilder()`.

Finally we call `checkSubsystemModelTransformation()` which reads the full legacy subsystem model. The legacy subsystem model will have been built up from the transformed boot operations from the parsed xml. The operations get transformed by the operation transformers. Then it takes the model of the current subsystem and transforms that using the resource transformers. Then it compares the two models, which should be the same. In some rare cases it is not possible to get those two models exactly the same, so there is a version of this method that takes a `ModelFixer` to make adjustments. The `checkSubsystemModelTransformation()` method also makes sure that the legacy model is valid according to the legacy subsystem's resource definition.

The legacy subsystem resource definitions are read on demand from the legacy controller when the tests run. In some older versions of subsystems (before we converted everything to use ResourceDefinition, and DescriptionProvider implementations were coded by hand) there were occasional problems with the resource definitions and they needed to be touched up. In this case you can generate a new one, touch it up and store the result in a file in the test resources under `/same/package/as/the/test/class/{{subsystem-name-model-version`. This will then prefer the file read from the file system to the one read at runtime. To generate the .dmr file, you need to generate it by adding a temporary test (make sure that you adjust `controllerVersion` and `modelVersion` to what you want to generate):

```
@Test
    public void deleteMeWhenDone() throws Exception {
        ModelTestControllerVersion controllerVersion = ModelTestControllerVersion.V7_1_2_FINAL;
        ModelVersion modelVersion = ModelVersion.create(1, 0, 0);
        KernelServicesBuilder builder = createKernelServicesBuilder(null);

        builder.createLegacyKernelServicesBuilder(null, controllerVersion, modelVersion)
            .addMavenResourceURL("org.jboss.as:jboss-as-threads:" +
controllerVersion.getMavenGavVersion());
        KernelServices services = builder.build();

        generateLegacySubsystemResourceRegistrationDmr(services, modelVersion);
    }
```

Now run the test and delete it. The legacy .dmr file should be in `target/test-classes/org/jboss/as/subsystem/test/<your-subsystem-name>-<your-vers`. Copy this .dmr file to the correct location in your project's test resources.

Testing a configuration that does not work

The `threads` subsystem (like several others) did not support the use of expression values in the version that came with JBoss AS 7.1.2.Final. So we have a test that attempts to use expressions, and then fixes each resource and attribute where expressions were not allowed.

```
@Test
    public void testRejectExpressionsAS712() throws Exception {
        testRejectExpressions_1_0_0(ModelTestControllerVersion.V7_1_2_FINAL);
    }

    private void testRejectExpressions_1_0_0(ModelTestControllerVersion controllerVersion)
throws Exception {
        // create builder for current subsystem version
        KernelServicesBuilder builder =
createKernelServicesBuilder(createAdditionalInitialization());

        // create builder for legacy subsystem version
        ModelVersion version_1_0_0 = ModelVersion.create(1, 0, 0);
        builder.createLegacyKernelServicesBuilder(null, controllerVersion, version_1_0_0)
                .addMavenResourceURL("org.jboss.as:jboss-as-threads:" +
controllerVersion.getMavenGavVersion())
                .excludeFromParent(SingleClassFilter.createFilter(ThreadsLogger.class));

        KernelServices mainServices = builder.build();
        KernelServices legacyServices = mainServices.getLegacyServices(version_1_0_0);

        Assert.assertNotNull(legacyServices);
        Assert.assertTrue("main services did not boot", mainServices.isSuccessfulBoot());
        Assert.assertTrue(legacyServices.isSuccessfulBoot());

        List<ModelNode> xmlOps = builder.parseXmlResource("expressions.xml");

        ModelTestUtils.checkFailedTransformedBootOperations(mainServices, version_1_0_0, xmlOps,
getConfig());
    }
```

Again we boot up a current and a legacy controller. However, note in this case that they are both empty, no xml was parsed on boot so there are no operations to boot up the model. Instead once the controllers have been booted, we call `KernelServicesBuilder.parseXmlResource()` which gets the operations from `expressions.xml`. `expressions.xml` uses expressions in all the places they were not allowed in 7.1.2.Final. For each resource `ModelTestUtils.checkFailedTransformedBootOperations()` will check that the `add` operation gets rejected, and then correct one attribute at a time until the resource has been totally corrected. Once the `add` operation is totally correct, it will check that the add operation no longer is rejected. The configuration for this is the `FailedOperationTransformationConfig` returned by the `getConfig()` method:

```
private FailedOperationTransformationConfig getConfig() {
        PathAddress subsystemAddress = PathAddress.pathAddress(ThreadsExtension.SUBSYSTEM_PATH);
        FailedOperationTransformationConfig.RejectExpressionsConfig allowedAndKeepalive =
                new
FailedOperationTransformationConfig.RejectExpressionsConfig(PoolAttributeDefinitions.ALLOW_CORE_TI
PoolAttributeDefinitions.KEEPALIVE_TIME);
...
        return new FailedOperationTransformationConfig()

.addFailedAttribute(subsystemAddress.append(PathElement.pathElement(CommonAttributes.BLOCKING_BOUN
allowedAndKeepalive)

.addFailedAttribute(subsystemAddress.append(PathElement.pathElement(CommonAttributes.BOUNDED_QUEUE
allowedAndKeepalive)
    }
```

So what this means is that we expect the `allow-core-timeout` and `keepalive-time` attributes for the `blocking-bounded-queue-thread-pool=*` and `bounded-queue-thread-pool=*` add operations to use expressions in the parsed xml. We then expect them to fail since there should be transformers in place to reject expressions, and correct them one at a time until the add operation should pass. As well as doing the `add` operations the `ModelTestUtils.checkFailedTransformedBootOperations()` method will also try calling `write-attribute` for each attribute, correcting as it goes along. As well as allowing you to test rejection of expressions `FailedOperationTransformationConfig` also has some helper classes to help testing rejection of other scenarios.

11.10.9 Common transformation use-cases

Most transformations are quite similar, so this section covers some of the actual transformation patterns found in the WildFly codebase. We will look at the output of CompareModelVersionsUtil, and see what can be done to transform for the older slave HCs. The examples come from the WildFly codebase but are stripped down to focus solely on the use-case being explained in an attempt to keep things as clear/simple as possible.

Child resource type does not exist in legacy model

Looking at the model comparison between WildFly and JBoss AS 7.2.0, there is a change to the `remoting` subsystem. The relevant part of the output is:

```
====== Resource root address: ["subsystem" => "remoting"] - Current version: 2.0.0; legacy
version: 1.2.0 =======
--- Problems for relative address to root []:
Missing child types in current: []; missing in legacy [http-connector]
```

So our current model has added a child type called `http-connector` which was not there in 7.2.0. This is configurable, and adds new behavior, so it can not be part of a configuration sent across to a legacy slave running version 1.2.0. So we add the following to `RemotingExtension` to reject all instances of that child type against ModelVersion 1.2.0.

```
@Override
    public void initialize(ExtensionContext context) {
        ....
        if (context.isRegisterTransformers()) {
            registerTransformers_1_1(registration);
            registerTransformers_1_2(registration);
        }
    }

    private void registerTransformers_1_2(SubsystemRegistration registration) {
        TransformationDescription.Tools.register(get1_2_0_1_3_0Description(), registration,
VERSION_1_2);
    }

    private static TransformationDescription get1_2_0_1_3_0Description() {
        ResourceTransformationDescriptionBuilder builder =
ResourceTransformationDescriptionBuilder.Factory.createSubsystemInstance();
        builder.rejectChildResource(HttpConnectorResource.PATH);

        return builder.build();
    }
```

Since this child resource type also does not exist in ModelVersion 1.1.0 we need to reject it there as well using a similar mechanism.

Attribute does not exist in the legacy subsystem

Default value of the attribute is the same as legacy implied behavior

This example also comes from the `remoting` subsystem, and is probably the most common type of transformation. The comparison tells us that there is now an attribute under `/subsystem=remoting/remote-outbound-connection=*` called `protocol` which did not exist in the older version:

```
====== Resource root address: ["subsystem" => "remoting"] - Current version: 2.0.0; legacy
version: 1.2.0 =======
--- Problems for relative address to root []:
....
--- Problems for relative address to root ["remote-outbound-connection" => "*"]:
Missing attributes in current: []; missing in legacy [protocol]
Missing parameters for operation 'add' in current: []; missing in legacy [protocol]
```

This difference also affects the `add` operation. Looking at the current model the valid values for the `protocol` attribute are `remote`, `http-remoting` and `https-remoting`. The last two are new protocols introduced in WildFly 8, meaning that the *implied behaviour* in JBoss 7.2.0 and earlier is the `remote` protocol. Since this attribute does not exist in the legacy model we want to discard this attribute if it is `undefined` or if it has the value `remote`, both of which are in line with what the legacy slave HC is hardwired to use. Also we want to reject it if it has a value different from `remote`. So what we need to do when registering transformers against ModelVersion 1.2.0 to handle this attribute:

```
private void registerTransformers_1_2(SubsystemRegistration registration) {
        TransformationDescription.Tools.register(get1_2_0_1_3_0Description(), registration,
VERSION_1_2);
    }

    private static TransformationDescription get1_2_0_1_3_0Description() {
        ResourceTransformationDescriptionBuilder builder =
ResourceTransformationDescriptionBuilder.Factory.createSubsystemInstance();

protocolTransform(builder.addChildResource(RemoteOutboundConnectionResourceDefinition.ADDRESS)
                .getAttributeBuilder());
        return builder.build();
    }

    private static AttributeTransformationDescriptionBuilder
protocolTransform(AttributeTransformationDescriptionBuilder builder) {
        builder.setDiscard(new DiscardAttributeChecker.DiscardAttributeValueChecker(new
ModelNode(Protocol.REMOTE.toString())), RemoteOutboundConnectionResourceDefinition.PROTOCOL)
                .addRejectCheck(RejectAttributeChecker.DEFINED,
RemoteOutboundConnectionResourceDefinition.PROTOCOL);
        return builder;
    }
```

So the first thing to happens is that we register a `DiscardAttributeChecker.DiscardAttributeValueChecker` which discards the attribute if it is either `undefined` (the default value in the current model is `remote`), or `defined` and has the value `remote`. Remembering that the `discard` phase always happens before the `reject` phase, the reject checker checks that the `protocol` attribute is defined, and rejects it if it is. The only reason it would be `defined` in the reject check, is if it was not discarded by the discard check. Hopefully this example shows that the discard and reject checkers often work in pairs.

An alternative way to write the `protocolTransform()` method would be:

```
private static AttributeTransformationDescriptionBuilder
protocolTransform(AttributeTransformationDescriptionBuilder builder) {
        builder.setDiscard(new DiscardAttributeChecker.DefaultDiscardAttributeChecker() {
                    @Override
                    protected boolean isValueDiscardable(final PathAddress address, final String
attributeName, final ModelNode attributeValue, final TransformationCon
                        return !attributeValue.isDefined() ||
attributeValue.asString().equals(Protocol.REMOTE.toString());
                    }
                }, RemoteOutboundConnectionResourceDefinition.PROTOCOL)
         .addRejectCheck(RejectAttributeChecker.DEFINED,
RemoteOutboundConnectionResourceDefinition.PROTOCOL);
         return builder;
```

The reject check remains the same, but we have implemented the discard check by using `DiscardAttributeChecker.DefaultDiscardAttributeChecker` instead. However, the effect of the discard check is exactly the same as when we used `DiscardAttributeChecker.DiscardAttributeValueChecker`.

Default value of the attribute is different from legacy implied behaviour

We touched on this in the weld subsystem example we used earlier in this guide, but let's take a more thorough look. Our comparison tells us that we have two new attributes `require-bean-descriptor` and `non-portable-mode`:

```
====== Resource root address: ["subsystem" => "weld"] - Current version: 2.0.0; legacy version:
1.0.0 =======
--- Problems for relative address to root []:
Missing attributes in current: []; missing in legacy [require-bean-descriptor,
non-portable-mode]
Missing parameters for operation 'add' in current: []; missing in legacy
[require-bean-descriptor, non-portable-mode]
```

Now when we look at this we see that the default value for both of the attributes in the current model is `false`, which allows us more flexible behavior introduced in CDI 1.1 (which was introduced with this version of the subsystem). The old model does not have these attributes, and implements CDI 1.0, which under the hood (using our weld subsystem expertise knowledge) implies the values `true` for both of these. So our transformer must reject anything that is not `true` for these attributes. Let us look at the transformer registered by the WeldExtension:

```
private void registerTransformers(SubsystemRegistration subsystem) {
        ResourceTransformationDescriptionBuilder builder =
TransformationDescriptionBuilder.Factory.createSubsystemInstance();
        //These new attributes are assumed to be 'true' in the old version but default to false
in the current version. So discard if 'true' and reject if 'undefined'.
        builder.getAttributeBuilder()
                .setDiscard(new DiscardAttributeChecker.DiscardAttributeValueChecker(false,
false, new ModelNode(true)),
                        WeldResourceDefinition.NON_PORTABLE_MODE_ATTRIBUTE,
WeldResourceDefinition.REQUIRE_BEAN_DESCRIPTOR_ATTRIBUTE)
                .addRejectCheck(new RejectAttributeChecker.DefaultRejectAttributeChecker() {

                    @Override
                    public String getRejectionLogMessage(Map<String, ModelNode> attributes) {
                        return
WeldMessages.MESSAGES.rejectAttributesMustBeTrue(attributes.keySet());
                    }

                    @Override
                    protected boolean rejectAttribute(PathAddress address, String attributeName,
ModelNode attributeValue,
                            TransformationContext context) {
                        //This will not get called if it was discarded, so reject if it is
undefined (default==false) or if defined and != 'true'
                        return !attributeValue.isDefined() ||
!attributeValue.asString().equals("true");
                    }
                }, WeldResourceDefinition.NON_PORTABLE_MODE_ATTRIBUTE,
WeldResourceDefinition.REQUIRE_BEAN_DESCRIPTOR_ATTRIBUTE)
                .end();
        TransformationDescription.Tools.register(builder.build(), subsystem,
ModelVersion.create(1, 0, 0));
    }
```

This looks a bit more scary than the previous transformer we have seen, but isn't actually too bad. The first thing we do is register a `DiscardAttributeChecker.DiscardAttributeValueChecker` which will discard the attribute if it has the value `true`. It will not discard if it is `undefined` since that defaults to `false`. This is registered for both attributes.

If the attributes had the value `true` they will get discarded we will not hit the reject checker since discarded attributes never get checked for rejection. If on the other hand they were an expression (since we are interested in the actual value, but cannot evaluate what value an expression will resolve to on the target from the DC running the transformers), `false`, or `undefined` (which will then default to `false`) they will not get discarded and will need to be rejected. So our `RejectAttributeChecker.DefaultRejectAttributeChecker.rejectAttribute()` method will return `true` (i.e. reject) if the attribute value is `undefined` (since that defaults to `false`) or if it is defined and 'not equal to `true`'. It is better to check for 'not equal to `true`' than to check for 'equal to `false`' since if an expression was used we still want to reject, and only the 'not equal to `true`' check would actually kick in in that case.

The other thing we need in our `DiscardAttributeChecker.DiscardAttributeValueChecker` is to override the `getRejectionLogMessage()` method to get the message to be displayed when rejecting the transformation. In this case it says something along the lines "These attributes must be 'true' for use with CDI 1.0 '%s'", with the names of the attributes having been rejected substituting the `%s`.

Attribute has a different default value

– TODO

(The gist of this is to use a value converter, such that if the attribute is undefined, and hence the default value will take effect, then the value gets converted to the current version's default value. This ensures that the legacy HC will use the same effective setting as current version HCs.

Note however that a change in default values is a form of incompatible API change, since CLI scripts written assuming the old defaults will now produce a configuration that behaves differently. Transformers make it possible to have a consistently configured domain even in the presence of this kind of incompatible change, but that doesn't mean such changes are good practice. They are generally unacceptable in WildFly's own subsystems.

One trick to ameliorate the impact of a default value change is to modify the xml parser for the **old** schema version such that if the xml attribute is not configured, the parser sets the old default value for the attribute, instead of `undefined`. This approach allows the parsing of old config documents to produce results consistent with what happened when they were created. It does not help with CLI scripts though.)

Attribute has a different type

Here the example comes from the `capacity` parameter some way into the `modcluster` subsystem, and the legacy version is AS 7.1.2.Final. There are quite a few differences, so I am only showing the ones relevant for this example:

```
====== Resource root address: ["subsystem" => "modcluster"] - Current version: 2.0.0; legacy
version: 1.2.0 =======
...
--- Problems for relative address to root ["mod-cluster-config" =>
"configuration","dynamic-load-provider" => "configuration","custom-load-m
etric" => "*"]:
Different 'type' for attribute 'capacity'. Current: DOUBLE; legacy: INT
Different 'expressions-allowed' for attribute 'capacity'. Current: true; legacy: false
...
Different 'type' for parameter 'capacity' of operation 'add'. Current: DOUBLE; legacy: INT
Different 'expressions-allowed' for parameter 'capacity' of operation 'add'. Current: true;
legacy: false
```

So as we can see expressions are not allowed for the `capacity` attribute, and the current type is `double` while the legacy subsystem is `int`. So this means that if the value is for example `2.0` we can convert this to `2`, but `2.5` cannot be converted. The way this is solved in the ModClusterExtension is to register the following some other attributes are registered here, but hopefully it is clear anyway:

```
dynamicLoadProvider.addChildResource(LOAD_METRIC_PATH)
                    .getAttributeBuilder()
                        .addRejectCheck(RejectAttributeChecker.SIMPLE_EXPRESSIONS, TYPE, WEIGHT,
CAPACITY, PROPERTY)
                        .addRejectCheck(CapacityCheckerAndConverter.INSTANCE, CAPACITY)
                        .setValueConverter(CapacityCheckerAndConverter.INSTANCE, CAPACITY)
                        ...
                        .end();
```

So we register that we should reject expressions, and we also register the `CapacityCheckerAndConverter` for `capacity`. `CapacityCheckerAndConverter` extends the convenience class `DefaultCheckersAndConverter` which implements the `DiscardAttributeChecker`, `RejectAttributeChecker`, and `AttributeConverter` interfaces. We have seen `DiscardAttributeChecker` and `RejectAttributeChecker` in previous examples. Since we now need to convert a value we need an instance of `AttributeConverter`.

```
static class CapacityCheckerAndConverter extends DefaultCheckersAndConverter {

        static final CapacityCheckerAndConverter INSTANCE = new CapacityCheckerAndConverter();
```

We should not discard so `isValueDiscardable()` from `DiscardAttributeChecker` always returns `false`:

```
@Override
        protected boolean isValueDiscardable(PathAddress address, String attributeName,
ModelNode attributeValue, TransformationContext context) {
            //Not used for discard
            return false;
        }

        @Override
        public String getRejectionLogMessage(Map<String, ModelNode> attributes) {
            return
ModClusterMessages.MESSAGES.capacityIsExpressionOrGreaterThanIntegerMaxValue(attributes.get(CAPACI
}
```

Now we check to see if we can convert the attribute to an `int` and reject if not. Note that if it is an expression, we have no idea what its value will resolve to on the target host, so we need to reject it. Then we try to change it into an `int`, and reject if that was not possible:

```
@Override
        protected boolean rejectAttribute(PathAddress address, String attributeName, ModelNode
attributeValue, TransformationContext context) {
            if (checkForExpression(attributeValue)
                    || (attributeValue.isDefined() &&
!isIntegerValue(attributeValue.asDouble()))) {
                return true;
            }
            Long converted = convert(attributeValue);
            return (converted != null && (converted > Integer.MAX_VALUE || converted <
Integer.MIN_VALUE));
        }
```

And then finally we do the conversion:

```
@Override
        protected void convertAttribute(PathAddress address, String attributeName, ModelNode
attributeValue, TransformationContext context) {
            Long converted = convert(attributeValue);
            if (converted != null && converted <= Integer.MAX_VALUE && converted >=
Integer.MIN_VALUE) {
                attributeValue.set((int)converted.longValue());
            }
        }


        private Long convert(ModelNode attributeValue) {
            if (attributeValue.isDefined() && !checkForExpression(attributeValue)) {
                double raw = attributeValue.asDouble();
                if (isIntegerValue(raw)) {
                    return Math.round(raw);
                }
            }
            return null;
        }

        private boolean isIntegerValue(double raw) {
            return raw == Double.valueOf(Math.round(raw)).doubleValue();
        }

    }
```

11.11 Example subsystem

Our example subsystem will keep track of all deployments of certain types containing a special marker file, and expose operations to see how long these deployments have been deployed.

11.11.1 Create the skeleton project

To make your life easier we have provided a maven archetype which will create a skeleton project for implementing subsystems.

```
mvn archetype:generate \
    -DarchetypeArtifactId=wildfly-subsystem \
    -DarchetypeGroupId=org.wildfly.archetypes \
    -DarchetypeVersion=8.0.0.Final \
    -DarchetypeRepository=http://repository.jboss.org/nexus/content/groups/public
```

Maven will download the archetype and it's dependencies, and ask you some questions:

```
$ mvn archetype:generate \
    -DarchetypeArtifactId=wildfly-subsystem \
    -DarchetypeGroupId=org.wildfly.archetypes \
    -DarchetypeVersion=8.0.0.Final \
    -DarchetypeRepository=http://repository.jboss.org/nexus/content/groups/public
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO] ------------------------------------------------------------------------
[INFO]

.........

Define value for property 'groupId': : com.acme.corp
Define value for property 'artifactId': : acme-subsystem
Define value for property 'version':  1.0-SNAPSHOT: :
Define value for property 'package':  com.acme.corp: : com.acme.corp.tracker
Define value for property 'module': : com.acme.corp.tracker
[INFO] Using property: name = WildFly subsystem project
Confirm properties configuration:
groupId: com.acme.corp
artifactId: acme-subsystem
version: 1.0-SNAPSHOT
package: com.acme.corp.tracker
module: com.acme.corp.tracker
name: WildFly subsystem project
 Y: : Y
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1:42.563s
[INFO] Finished at: Fri Jul 08 14:30:09 BST 2011
[INFO] Final Memory: 7M/81M
[INFO] ------------------------------------------------------------------------
$
```

	Instruction
1	Enter the `groupId` you wish to use
2	Enter the `artifactId` you wish to use
3	Enter the version you wish to use, or just hit `Enter` if you wish to accept the default `1.0-SNAPSHOT`
4	Enter the java package you wish to use, or just hit `Enter` if you wish to accept the default (which is copied from `groupId`).
5	Enter the module name you wish to use for your extension.
6	Finally, if you are happy with your choices, hit `Enter` and Maven will generate the project for you.

You can also do this in Eclipse, see Creating your own application for more details. We now have a skeleton project that you can use to implement a subsystem. Import the `acme-subsystem` project into your favourite IDE. A nice side-effect of running this in the IDE is that you can see the javadoc of WildFly classes and interfaces imported by the skeleton code. If you do a `mvn install` in the project it will work if we plug it into WildFly, but before doing that we will change it to do something more useful.

The rest of this section modifies the skeleton project created by the archetype to do something more useful, and the full code can be found in acme-subsystem.zip.

If you do a `mvn install` in the created project, you will see some tests being run

```
$mvn install
[INFO] Scanning for projects...
[...]
[INFO] Surefire report directory:
/Users/kabir/sourcecontrol/temp/archetype-test/acme-subsystem/target/surefire-reports

-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running com.acme.corp.tracker.extension.SubsystemBaseParsingTestCase
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.424 sec
Running com.acme.corp.tracker.extension.SubsystemParsingTestCase
Tests run: 6, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.074 sec

Results :

Tests run: 3, Failures: 0, Errors: 0, Skipped: 0
[...]
```

We will talk about these later in the Testing the parsers section.

11.11.2 Create the schema

First, let us define the schema for our subsystem. Rename `src/main/resources/schema/mysubsystem.xsd` to `src/main/resources/schema/acme.xsd`. Then open `acme.xsd` and modify it to the following

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
            targetNamespace="urn:com.acme.corp.tracker:1.0"
            xmlns="urn:com.acme.corp.tracker:1.0"
            elementFormDefault="qualified"
            attributeFormDefault="unqualified"
            version="1.0">

   <!-- The subsystem root element -->
   <xs:element name="subsystem" type="subsystemType"/>
   <xs:complexType name="subsystemType">
      <xs:all>
         <xs:element name="deployment-types" type="deployment-typesType"/>
      </xs:all>
   </xs:complexType>
   <xs:complexType name="deployment-typesType">
      <xs:choice minOccurs="0" maxOccurs="unbounded">
         <xs:element name="deployment-type" type="deployment-typeType"/>
      </xs:choice>
   </xs:complexType>
   <xs:complexType name="deployment-typeType">
      <xs:attribute name="suffix" use="required"/>
      <xs:attribute name="tick" type="xs:long" use="optional" default="10000"/>
   </xs:complexType>
</xs:schema>
```

Note that we modified the `xmlns` and `targetNamespace` values to `urn.com.acme.corp.tracker:1.0`. Our new `subsystem` element has a child called `deployment-types`, which in turn can have zero or more children called `deployment-type`. Each `deployment-type` has a required `suffix` attribute, and a `tick` attribute which defaults to `true`.
Now modify the `com.acme.corp.tracker.extension.SubsystemExtension` class to contain the new namespace.

```
public class SubsystemExtension implements Extension {

    /** The name space used for the {@code substystem} element */
    public static final String NAMESPACE = "urn:com.acme.corp.tracker:1.0";
    ...
```

11.11.3 Design and define the model structure

The following example xml contains a valid subsystem configuration, we will see how to plug this in to WildFly later in this tutorial.

```
<subsystem xmlns="urn:com.acme.corp.tracker:1.0">
   <deployment-types>
      <deployment-type suffix="sar" tick="10000"/>
      <deployment-type suffix="war" tick="10000"/>
   </deployment-types>
</subsystem>
```

Now when designing our model, we can either do a one to one mapping between the schema and the model or come up with something slightly or very different. To keep things simple, let us stay pretty true to the schema so that when executing a `:read-resource(recursive=true)` against our subsystem we'll see something like:

```
{
    "outcome" => "success",
    "result" => {"type" => {
        "sar" => {"tick" => "10000"},
        "war" => {"tick" => "10000"}
    }}
}
```

Each `deployment-type` in the xml becomes in the model a child resource of the subsystem's root resource. The child resource's child-type is `type`, and it is indexed by its `suffix`. Each `type` resource then contains the `tick` attribute.
We also need a name for our subsystem, to do that change `com.acme.corp.tracker.extension.SubsystemExtension`:

```
public class SubsystemExtension implements Extension {
    ...
    /** The name of our subsystem within the model. */
    public static final String SUBSYSTEM_NAME = "tracker";
    ...
```

Once we are finished our subsystem will be available under `/subsystem=tracker`.
The `SubsystemExtension.initialize()` method defines the model, currently it sets up the basics to add our subsystem to the model:

```
@Override
    public void initialize(ExtensionContext context) {
        //register subsystem with its model version
        final SubsystemRegistration subsystem = context.registerSubsystem(SUBSYSTEM_NAME, 1, 0);
        //register subsystem model with subsystem definition that defines all attributes and
operations
        final ManagementResourceRegistration registration =
subsystem.registerSubsystemModel(SubsystemDefinition.INSTANCE);
        //register describe operation, note that this can be also registered in
SubsystemDefinition
        registration.registerOperationHandler(DESCRIBE,
GenericSubsystemDescribeHandler.INSTANCE, GenericSubsystemDescribeHandler.INSTANCE, false,
OperationEntry.EntryType.PRIVATE);
        //we can register additional submodels here
        //
        subsystem.registerXMLElementWriter(parser);
    }
```

The `registerSubsystem()` call registers our subsystem with the extension context. At the end of the method we register our parser with the returned `SubsystemRegistration` to be able to marshal our subsystem's model back to the main configuration file when it is modified. We will add more functionality to this method later.

Registering the core subsystem model

Next we obtain a `ManagementResourceRegistration` by registering the subsystem model. This is a **compulsory** step for every new subsystem.

```
final ManagementResourceRegistration registration =
subsystem.registerSubsystemModel(SubsystemDefinition.INSTANCE);
```

Its parameter is an implementation of the `ResourceDefinition` interface, which means that when you call `/subsystem=tracker:read-resource-description` the information you see comes from model that is defined by `SubsystemDefinition.INSTANCE`.

```
public class SubsystemDefinition extends SimpleResourceDefinition {
    public static final SubsystemDefinition INSTANCE = new SubsystemDefinition();

    private SubsystemDefinition() {
        super(SubsystemExtension.SUBSYSTEM_PATH,
                SubsystemExtension.getResourceDescriptionResolver(null),
                //We always need to add an 'add' operation
                SubsystemAdd.INSTANCE,
                //Every resource that is added, normally needs a remove operation
                SubsystemRemove.INSTANCE);
    }

    @Override
    public void registerOperations(ManagementResourceRegistration resourceRegistration) {
        super.registerOperations(resourceRegistration);
        //you can register aditional operations here
    }

    @Override
    public void registerAttributes(ManagementResourceRegistration resourceRegistration) {
        //you can register attributes here
    }
}
```

Since we need child resource `type` we need to add new ResourceDefinition,

The `ManagementResourceRegistration` obtained in `SubsystemExtension.initialize()` is then used to add additional operations or to register submodels to the `/subsystem=tracker` address. Every subsystem and resource **must** have an `ADD` method which can be achieved by the following line inside `registerOperations` in your `ResourceDefinition` or by providing it in constructor of your `SimpleResourceDefinition` just as we did in example above.

```
//We always need to add an 'add' operation
        resourceRegistration.registerOperationHandler(ADD, SubsystemAdd.INSTANCE, new
DefaultResourceAddDescriptionProvider(resourceRegistration,descriptionResolver), false);
```

The parameters when registering an operation handler are:

1. **The name** - i.e. `ADD`.
2. The handler instance - we will talk more about this below
3. The handler description provider - we will talk more about this below.
4. Whether this operation handler is inherited - `false` means that this operation is not inherited, and will only apply to `/subsystem=tracker`. The content for this operation handler will be provided by 3.

Let us first look at the description provider which is quite simple since this operation takes no parameters. The addition of `type` children will be handled by another operation handler, as we will see later on.

There are two way to define `DescriptionProvider`, one is by defining it by hand using ModelNode, but as this has show to be very error prone there are lots of helper methods to help you automatically describe the model. Following example is done by manually defining Description provider for ADD operation handler

```
/**
     * Used to create the description of the subsystem add method
     */
    public static DescriptionProvider SUBSYSTEM_ADD = new DescriptionProvider() {
        public ModelNode getModelDescription(Locale locale) {
            //The locale is passed in so you can internationalize the strings used in the
descriptions

            final ModelNode subsystem = new ModelNode();
            subsystem.get(OPERATION_NAME).set(ADD);
            subsystem.get(DESCRIPTION).set("Adds the tracker subsystem");

            return subsystem;
        }
    };
```

Or you can use API that helps you do that for you. For Add and Remove methods there are classes `DefaultResourceAddDescriptionProvider` and `DefaultResourceRemoveDescriptionProvider` that do work for you. In case you use `SimpleResourceDefinition` even that part is hidden from you.

```
resourceRegistration.registerOperationHandler(ADD, SubsystemAdd.INSTANCE, new
DefaultResourceAddDescriptionProvider(resourceRegistration,descriptionResolver), false);
resourceRegistration.registerOperationHandler(REMOVE, SubsystemRemove.INSTANCE, new
DefaultResourceRemoveDescriptionProvider(resourceRegistration,descriptionResolver), false);
```

For other operation handlers that are not add/remove you can use `DefaultOperationDescriptionProvider` that takes additional parameter of what is the name of operation and optional array of parameters/attributes operation takes. This is an example to register operation "`add-mime`" with two parameters:

```
container.registerOperationHandler("add-mime",
                MimeMappingAdd.INSTANCE,
                new DefaultOperationDescriptionProvider("add-mime",
Extension.getResourceDescriptionResolver("container.mime-mapping"), MIME_NAME, MIME_VALUE));
```

When descriping an operation its description provider's `OPERATION_NAME` must match the name used when calling `ManagementResourceRegistration.registerOperationHandler()`

Next we have the actual operation handler instance, note that we have changed its `populateModel()` method to initialize the `type` child of the model.

```
class SubsystemAdd extends AbstractBoottimeAddStepHandler {

    static final SubsystemAdd INSTANCE = new SubsystemAdd();

    private SubsystemAdd() {
    }

    /** {@inheritDoc} */
    @Override
    protected void populateModel(ModelNode operation, ModelNode model) throws
OperationFailedException {
        log.info("Populating the model");
        //Initialize the 'type' child node
        model.get("type").setEmptyObject();
    }
    ....
```

`SubsystemAdd` also has a `performBoottime()` method which is used for initializing the deployer chain associated with this subsystem. We will talk about the deployers later on. However, the basic idea for all operation handlers is that we do any model updates before changing the actual runtime state.

The rule of thumb is that every thing that can be added, can also be removed so we have a remove handler for the subsystem registered
in `SubsystemDefinition.registerOperations` or just provide the operation handler in constructor.

```
//Every resource that is added, normally needs a remove operation
        registration.registerOperationHandler(REMOVE, SubsystemRemove.INSTANCE,
DefaultResourceRemoveDescriptionProvider(resourceRegistration,descriptionResolver) , false);
```

`SubsystemRemove` extends `AbstractRemoveStepHandler` which takes care of removing the resource from the model so we don't need to override its `performRemove()` operation, also the add handler did not install any services (services will be discussed later) so we can delete the `performRuntime()` method generated by the archetype.

```
class SubsystemRemove extends AbstractRemoveStepHandler {

    static final SubsystemRemove INSTANCE = new SubsystemRemove();

    private final Logger log = Logger.getLogger(SubsystemRemove.class);

    private SubsystemRemove() {
    }
}
```

The description provider for the remove operation is simple and quite similar to that of the add handler where just name of the method changes.

Registering the subsystem child

The `type` child does not exist in our skeleton project so we need to implement the operations to add and remove them from the model.

First we need an add operation to add the `type` child, create a class called `com.acme.corp.tracker.extension.TypeAddHandler`. In this case we extend the `org.jboss.as.controller.AbstractAddStepHandler` class and implement the `org.jboss.as.controller.descriptions.DescriptionProvider` interface. `org.jboss.as.controller.OperationStepHandler` is the main interface for the operation handlers, and `AbstractAddStepHandler` is an implementation of that which does the plumbing work for adding a resource to the model.

```
class TypeAddHandler extends AbstractAddStepHandler implements DescriptionProvider {

    public static final TypeAddHandler INSTANCE = new TypeAddHandler();

    private TypeAddHandler() {
    }
```

Then we define subsystem model. Lets call it `TypeDefinition` and for ease of use let it extend `SimpleResourceDefinition` instead just implement `ResourceDefinition`.

```
public class TypeDefinition extends SimpleResourceDefinition {

 public static final TypeDefinition INSTANCE = new TypeDefinition();

 //we define attribute named tick
protected static final SimpleAttributeDefinition TICK =
new SimpleAttributeDefinitionBuilder(TrackerExtension.TICK, ModelType.LONG)
  .setAllowExpression(true)
  .setXmlName(TrackerExtension.TICK)
  .setFlags(AttributeAccess.Flag.RESTART_ALL_SERVICES)
  .setDefaultValue(new ModelNode(1000))
  .setAllowNull(false)
  .build();

private TypeDefinition(){
   super(TYPE_PATH,
TrackerExtension.getResourceDescriptionResolver(TYPE),TypeAdd.INSTANCE,TypeRemove.INSTANCE);
}

@Override
public void registerAttributes(ManagementResourceRegistration resourceRegistration){
   resourceRegistration.registerReadWriteAttribute(TICK, null, TrackerTickHandler.INSTANCE);
}

}
```

Which will take care of describing the model for us. As you can see in example above we define `SimpleAttributeDefinition` named `TICK`, this is a mechanism to define Attributes in more type safe way and to add more common API to manipulate attributes. As you can see here we define default value of 1000 as also other constraints and capabilities. There could be other properties set such as validators, alternate names, xml name, flags for marking it attribute allows expressions and more.
Then we do the work of updating the model by implementing the `populateModel()` method from the `AbstractAddStepHandler`, which populates the model's attribute from the operation parameters. First we get hold of the model relative to the address of this operation (we will see later that we will register it against `/subsystem=tracker/type=*`), so we just specify an empty relative address, and we then populate our model with the parameters from the operation. There is operation `validateAndSet` on `AttributeDefinition` that helps us validate and set the model based on definition of the attribute.

```
@Override
    protected void populateModel(ModelNode operation, ModelNode model) throws
OperationFailedException {
         TICK.validateAndSet(operation,model);
    }
```

We then override the `performRuntime()` method to perform our runtime changes, which in this case involves installing a service into the controller at the heart of WildFly. (`AbstractAddStepHandler.performRuntime()` is similar to `AbstractBoottimeAddStepHandler.performBoottime()` in that the model is updated before runtime changes are made.

```
@Override
    protected void performRuntime(OperationContext context, ModelNode operation, ModelNode
model,
            ServiceVerificationHandler verificationHandler, List<ServiceController<?>>
newControllers)
            throws OperationFailedException {
        String suffix =
PathAddress.pathAddress(operation.get(ModelDescriptionConstants.ADDRESS)).getLastElement().getValu
long tick = TICK.resolveModelAttribute(context,model).asLong();
        TrackerService service = new TrackerService(suffix, tick);
        ServiceName name = TrackerService.createServiceName(suffix);
        ServiceController<TrackerService> controller = context.getServiceTarget()
                .addService(name, service)
                .addListener(verificationHandler)
                .setInitialMode(Mode.ACTIVE)
                .install();
        newControllers.add(controller);
    }
}
```

Since the add methods will be of the format `/subsystem=tracker/suffix=war:add(tick=1234)`, we look for the last element of the operation address, which is `war` in the example just given and use that as our suffix. We then create an instance of TrackerService and install that into the `service target` of the context and add the created `service controller` to the `newControllers` list.

The tracker service is quite simple. All services installed into WildFly must implement the `org.jboss.msc.service.Service` interface.

```
public class TrackerService implements Service<TrackerService>{
```

We then have some fields to keep the tick count and a thread which when run outputs all the deployments registered with our service.

```
private AtomicLong tick = new AtomicLong(10000);

    private Set<String> deployments = Collections.synchronizedSet(new HashSet<String>());
    private Set<String> coolDeployments = Collections.synchronizedSet(new HashSet<String>());
    private final String suffix;

    private Thread OUTPUT = new Thread() {
        @Override
        public void run() {
            while (true) {
                try {
                    Thread.sleep(tick.get());
                    System.out.println("Current deployments deployed while " + suffix + "
tracking active:\n" + deployments
                       + "\nCool: " + coolDeployments.size());
                } catch (InterruptedException e) {
                    interrupted();
                    break;
                }
            }
        }
    };

    public TrackerService(String suffix, long tick) {
        this.suffix = suffix;
        this.tick.set(tick);
    }
```

Next we have three methods which come from the `Service` interface. `getValue()` returns this service, `start()` is called when the service is started by the controller, `stop` is called when the service is stopped by the controller, and they start and stop the thread outputting the deployments.

```
@Override
    public TrackerService getValue() throws IllegalStateException, IllegalArgumentException {
        return this;
    }

    @Override
    public void start(StartContext context) throws StartException {
        OUTPUT.start();
    }

    @Override
    public void stop(StopContext context) {
        OUTPUT.interrupt();
    }
```

Next we have a utility method to create the `ServiceName` which is used to register the service in the controller.

```
public static ServiceName createServiceName(String suffix) {
        return ServiceName.JBOSS.append("tracker", suffix);
}
```

Finally we have some methods to add and remove deployments, and to set and read the `tick`. The 'cool' deployments will be explained later.

```
public void addDeployment(String name) {
        deployments.add(name);
    }

    public void addCoolDeployment(String name) {
        coolDeployments.add(name);
    }

    public void removeDeployment(String name) {
        deployments.remove(name);
        coolDeployments.remove(name);
    }

    void setTick(long tick) {
        this.tick.set(tick);
    }

    public long getTick() {
        return this.tick.get();
    }
}//TrackerService - end
```

Since we are able to add `type` children, we need a way to be able to remove them, so we create a `com.acme.corp.tracker.extension.TypeRemoveHandler`. In this case we extend `AbstractRemoveStepHandler` which takes care of removing the resource from the model so we don't need to override its `performRemove()` operationa. But we need to implement the `DescriptionProvider` method to provide the model description, and since the add handler installs the TrackerService, we need to remove that in the `performRuntime()` method.

```
public class TypeRemoveHandler extends AbstractRemoveStepHandler {

    public static final TypeRemoveHandler INSTANCE = new TypeRemoveHandler();

    private TypeRemoveHandler() {
    }


    @Override
    protected void performRuntime(OperationContext context, ModelNode operation, ModelNode
model) throws OperationFailedException {
        String suffix =
PathAddress.pathAddress(operation.get(ModelDescriptionConstants.ADDRESS)).getLastElement().getValu
ServiceName name = TrackerService.createServiceName(suffix);
        context.removeService(name);
    }

}
```

We then need a description provider for the `type` part of the model itself, so we modify TypeDefinitnion to registerAttribute

```
class TypeDefinition{
...
@Override
public void registerAttributes(ManagementResourceRegistration resourceRegistration){
    resourceRegistration.registerReadWriteAttribute(TICK, null, TrackerTickHandler.INSTANCE);
}

}
```

Then finally we need to specify that our new `type` child and associated handlers go under `/subsystem=tracker/type=*` in the model by adding registering it with the model in `SubsystemExtension.initialize()`. So we add the following just before the end of the method.

```
@Override
public void initialize(ExtensionContext context)
{
 final SubsystemRegistration subsystem = context.registerSubsystem(SUBSYSTEM_NAME, 1, 0);
 final ManagementResourceRegistration registration =
subsystem.registerSubsystemModel(TrackerSubsystemDefinition.INSTANCE);
 //Add the type child
 ManagementResourceRegistration typeChild =
registration.registerSubModel(TypeDefinition.INSTANCE);
 subsystem.registerXMLElementWriter(parser);
}
```

The above first creates a child of our main subsystem registration for the relative address `type=*`, and gets the `typeChild` registration.
To this we add the `TypeAddHandler` and `TypeRemoveHandler`.
The add variety is added under the name `add` and the remove handler under the name `remove`, and for each registered operation handler we use the handler singleton instance as both the handler parameter and as the `DescriptionProvider`.

Finally, we register `tick` as a read/write attribute, the null parameter means we don't do anything special with regards to reading it, for the write handler we supply it with an operation handler called `TrackerTickHandler`.
Registering it as a read/write attribute means we can use the `:write-attribute` operation to modify the value of the parameter, and it will be handled by `TrackerTickHandler`.

Not registering a write attribute handler makes the attribute read only.

`TrackerTickHandler` extends `AbstractWriteAttributeHandler`
directly, and so must implement its `applyUpdateToRuntime` and `revertUpdateToRuntime` method.
This takes care of model manipulation (validation, setting) but leaves us to do just to deal with what we need to do.

```
class TrackerTickHandler extends AbstractWriteAttributeHandler<Void> {

    public static final TrackerTickHandler INSTANCE = new TrackerTickHandler();

    private TrackerTickHandler() {
        super(TypeDefinition.TICK);
    }

    protected boolean applyUpdateToRuntime(OperationContext context, ModelNode operation, String
attributeName,
              ModelNode resolvedValue, ModelNode currentValue, HandbackHolder<Void>
handbackHolder) throws OperationFailedException {

        modifyTick(context, operation, resolvedValue.asLong());

        return false;
    }

    protected void revertUpdateToRuntime(OperationContext context, ModelNode operation, String
attributeName, ModelNode valueToRestore, ModelNode valueToRevert, Void handback){
        modifyTick(context, operation, valueToRestore.asLong());
    }

    private void modifyTick(OperationContext context, ModelNode operation, long value) throws
OperationFailedException {

        final String suffix =
PathAddress.pathAddress(operation.get(ModelDescriptionConstants.ADDRESS)).getLastElement().getValu
TrackerService service = (TrackerService)
context.getServiceRegistry(true).getRequiredService(TrackerService.createServiceName(suffix)).getV
service.setTick(value);
    }

}
```

The operation used to execute this will be of the form `/subsystem=tracker/type=war:write-attribute(name=tick,value=12345)` so we first get the `suffix` from the operation address, and the `tick` value from the operation parameter's `resolvedValue` parameter, and use that to update the model.

We then add a new step associated with the `RUNTIME` stage to update the tick of the TrackerService for our suffix. This is essential since the call to `context.getServiceRegistry()` will fail unless the step accessing it belongs to the `RUNTIME` stage.

 When implementing `execute()`, you **must** call `context.completeStep()` when you are done.

11.11.4 Parsing and marshalling of the subsystem xml

WildFly uses the Stax API to parse the xml files. This is initialized in `SubsystemExtension` by mapping our parser onto our namespace:

```
public class SubsystemExtension implements Extension {

    /** The name space used for the {@code subsystem} element */
    public static final String NAMESPACE = "urn:com.acme.corp.tracker:1.0";
    ...
    protected static final PathElement SUBSYSTEM_PATH = PathElement.pathElement(SUBSYSTEM,
SUBSYSTEM_NAME);
    protected static final PathElement TYPE_PATH = PathElement.pathElement(TYPE);

   /** The parser used for parsing our subsystem */
    private final SubsystemParser parser = new SubsystemParser();

   @Override
    public void initializeParsers(ExtensionParsingContext context) {
        context.setSubsystemXmlMapping(NAMESPACE, parser);
    }
    ...
```

We then need to write the parser. The contract is that we read our subsystem's xml and create the operations that will populate the model with the state contained in the xml. These operations will then be executed on our behalf as part of the parsing process. The entry point is the `readElement()` method.

```
public class SubsystemExtension implements Extension {

    /**
     * The subsystem parser, which uses stax to read and write to and from xml
     */
    private static class SubsystemParser implements XMLStreamConstants,
XMLElementReader<List<ModelNode>>, XMLElementWriter<SubsystemMarshallingContext> {

        /** {@inheritDoc} */
        @Override
        public void readElement(XMLExtendedStreamReader reader, List<ModelNode> list) throws
XMLStreamException {
            // Require no attributes
            ParseUtils.requireNoAttributes(reader);

            //Add the main subsystem 'add' operation
            final ModelNode subsystem = new ModelNode();
            subsystem.get(OP).set(ADD);
            subsystem.get(OP_ADDR).set(PathAddress.pathAddress(SUBSYSTEM_PATH).toModelNode());
            list.add(subsystem);

            //Read the children
            while (reader.hasNext() && reader.nextTag() != END_ELEMENT) {
                if (!reader.getLocalName().equals("deployment-types")) {
```

```
                    throw ParseUtils.unexpectedElement(reader);
                }
                while (reader.hasNext() && reader.nextTag() != END_ELEMENT) {
                    if (reader.isStartElement()) {
                        readDeploymentType(reader, list);
                    }
                }
            }
        }

        private void readDeploymentType(XMLExtendedStreamReader reader, List<ModelNode> list)
throws XMLStreamException {
            if (!reader.getLocalName().equals("deployment-type")) {
                throw ParseUtils.unexpectedElement(reader);
            }
            ModelNode addTypeOperation = new ModelNode();
            addTypeOperation.get(OP).set(ModelDescriptionConstants.ADD);

            String suffix = null;
            for (int i = 0; i < reader.getAttributeCount(); i++) {
                String attr = reader.getAttributeLocalName(i);
                String value = reader.getAttributeValue(i);
                if (attr.equals("tick")) {
                    TypeDefinition.TICK.parseAndSetParameter(value, addTypeOperation, reader);
                } else if (attr.equals("suffix")) {
                    suffix = value;
                } else {
                    throw ParseUtils.unexpectedAttribute(reader, i);
                }
            }
            ParseUtils.requireNoContent(reader);
            if (suffix == null) {
                throw ParseUtils.missingRequiredElement(reader,
Collections.singleton("suffix"));
            }

            //Add the 'add' operation for each 'type' child
            PathAddress addr = PathAddress.pathAddress(SUBSYSTEM_PATH,
PathElement.pathElement(TYPE, suffix));
            addTypeOperation.get(OP_ADDR).set(addr.toModelNode());
            list.add(addTypeOperation);
        }
        ...
```

So in the above we always create the add operation for our subsystem. Due to its address `/subsystem=tracker` defined by SUBSYSTEM_PATH this will trigger the `SubsystemAddHandler` we created earlier when we invoke `/subsystem=tracker:add`. We then parse the child elements and create an add operation for the child address for each `type` child. Since the address will for example be `/subsystem=tracker/type=sar` (defined by TYPE_PATH) and `TypeAddHandler` is registered for all `type` subaddresses the `TypeAddHandler` will get invoked for those operations. Note that when we are parsing attribute `tick` we are using definition of attribute that we defined in TypeDefintion to parse attribute value and apply all rules that we specified for this attribute, this also enables us to property support expressions on attributes.

The parser is also used to marshal the model to xml whenever something modifies the model, for which the entry point is the `writeContent()` method:

```
private static class SubsystemParser implements XMLStreamConstants,
XMLElementReader<List<ModelNode>>, XMLElementWriter<SubsystemMarshallingContext> {
        ...
        /** {@inheritDoc} */
        @Override
        public void writeContent(final XMLExtendedStreamWriter writer, final
SubsystemMarshallingContext context) throws XMLStreamException {
            //Write out the main subsystem element
            context.startSubsystemElement(TrackerExtension.NAMESPACE, false);
            writer.writeStartElement("deployment-types");
            ModelNode node = context.getModelNode();
            ModelNode type = node.get(TYPE);
            for (Property property : type.asPropertyList()) {

                //write each child element to xml
                writer.writeStartElement("deployment-type");
                writer.writeAttribute("suffix", property.getName());
                ModelNode entry = property.getValue();
                TypeDefinition.TICK.marshallAsAttribute(entry, true, writer);
                writer.writeEndElement();
            }
            //End deployment-types
            writer.writeEndElement();
            //End subsystem
            writer.writeEndElement();
        }
    }
```

Then we have to implement the `SubsystemDescribeHandler` which translates the current state of the model into operations similar to the ones created by the parser. The `SubsystemDescribeHandler` is only used when running in a managed domain, and is used when the host controller queries the domain controller for the configuration of the profile used to start up each server. In our case the `SubsystemDescribeHandler` adds the operation to add the subsystem and then adds the operation to add each `type` child. Since we are using ResourceDefinitinon for defining subsystem all that is generated for us, but if you want to customize that you can do it by implementing it like this.

```
private static class SubsystemDescribeHandler implements OperationStepHandler,
DescriptionProvider {
        static final SubsystemDescribeHandler INSTANCE = new SubsystemDescribeHandler();

        public void execute(OperationContext context, ModelNode operation) throws
OperationFailedException {
            //Add the main operation
            context.getResult().add(createAddSubsystemOperation());

            //Add the operations to create each child

            ModelNode node = context.readModel(PathAddress.EMPTY_ADDRESS);
            for (Property property : node.get("type").asPropertyList()) {

                ModelNode addType = new ModelNode();
                addType.get(OP).set(ModelDescriptionConstants.ADD);
                PathAddress addr = PathAddress.pathAddress(SUBSYSTEM_PATH,
PathElement.pathElement("type", property.getName()));
                addType.get(OP_ADDR).set(addr.toModelNode());
                if (property.getValue().hasDefined("tick")) {
                   TypeDefinition.TICK.validateAndSet(property,addType);
                }
                context.getResult().add(addType);
            }
            context.completeStep();
        }


}
```

Testing the parsers

Changes to tests between 7.0.0 and 7.0.1

The testing framework was moved from the archetype into the core JBoss AS 7 sources between JBoss AS 7.0.0 and JBoss AS 7.0.1, and has been improved upon and is used internally for testing JBoss AS 7's subsystems. The differences between the two versions is that in 7.0.0.Final the testing framework is bundled with the code generated by the archetype (in a sub-package of the package specified for your subsystem, e.g. `com.acme.corp.tracker.support`), and the test extends the `AbstractParsingTest` class.

From 7.0.1 the testing framework is now brought in via the `org.jboss.as:jboss-as-subsystem-test` maven artifact, and the test's superclass is `org.jboss.as.subsystem.test.AbstractSubsystemTest`. The concepts are the same but more and more functionality will be available as JBoss AS 7 is developed.

Now that we have modified our parsers we need to update our tests to reflect the new model. There are currently three tests testing the basic functionality, something which is a lot easier to debug from your IDE before you plug it into the application server. We will talk about these tests in turn and they all live in `com.acme.corp.tracker.extension.SubsystemParsingTestCase`. `SubsystemParsingTestCase` extends `AbstractSubsystemTest` which does a lot of the setup for you and contains utility methods for verifying things from your test. See the javadoc of that class for more information about the functionality available to you. And by all means feel free to add more tests for your subsystem, here we are only testing for the best case scenario while you will probably want to throw in a few tests for edge cases.

The first test we need to modify is `testParseSubsystem()`. It tests that the parsed xml becomes the expected operations that will be parsed into the server, so let us tweak this test to match our subsystem. First we tell the test to parse the xml into operations

```
@Test
    public void testParseSubsystem() throws Exception {
        //Parse the subsystem xml into operations
        String subsystemXml =
                "<subsystem xmlns=\"" + SubsystemExtension.NAMESPACE + "\">" +
                "   <deployment-types>" +
                "       <deployment-type suffix=\"tst\" tick=\"12345\"/>" +
                "   </deployment-types>" +
                "</subsystem>";
        List<ModelNode> operations = super.parse(subsystemXml);
```

There should be one operation for adding the subsystem itself and an operation for adding the `deployment-type`, so check we got two operations

```
///Check that we have the expected number of operations
        Assert.assertEquals(2, operations.size());
```

Now check that the first operation is `add` for the address `/subsystem=tracker`:

```
//Check that each operation has the correct content
        //The add subsystem operation will happen first
        ModelNode addSubsystem = operations.get(0);
        Assert.assertEquals(ADD, addSubsystem.get(OP).asString());
        PathAddress addr = PathAddress.pathAddress(addSubsystem.get(OP_ADDR));
        Assert.assertEquals(1, addr.size());
        PathElement element = addr.getElement(0);
        Assert.assertEquals(SUBSYSTEM, element.getKey());
        Assert.assertEquals(SubsystemExtension.SUBSYSTEM_NAME, element.getValue());
```

Then check that the second operation is `add` for the address `/subsystem=tracker`, and that `12345` was picked up for the value of the `tick` parameter:

```
//Then we will get the add type operation
        ModelNode addType = operations.get(1);
        Assert.assertEquals(ADD, addType.get(OP).asString());
        Assert.assertEquals(12345, addType.get("tick").asLong());
        addr = PathAddress.pathAddress(addType.get(OP_ADDR));
        Assert.assertEquals(2, addr.size());
        element = addr.getElement(0);
        Assert.assertEquals(SUBSYSTEM, element.getKey());
        Assert.assertEquals(SubsystemExtension.SUBSYSTEM_NAME, element.getValue());
        element = addr.getElement(1);
        Assert.assertEquals("type", element.getKey());
        Assert.assertEquals("tst", element.getValue());
    }
```

The second test we need to modify is `testInstallIntoController()` which tests that the xml installs properly into the controller. In other words we are making sure that the `add` operations we created earlier work properly. First we create the xml and install it into the controller. Behind the scenes this will parse the xml into operations as we saw in the last test, but it will also create a new controller and boot that up using the created operations

```
@Test
    public void testInstallIntoController() throws Exception {
        //Parse the subsystem xml and install into the controller
        String subsystemXml =
                "<subsystem xmlns=\"" + SubsystemExtension.NAMESPACE + "\">" +
                "   <deployment-types>" +
                "       <deployment-type suffix=\"tst\" tick=\"12345\"/>" +
                "   </deployment-types>" +
                "</subsystem>";
        KernelServices services = super.installInController(subsystemXml);
```

The returned `KernelServices` allow us to execute operations on the controller, and to read the whole model.

```
//Read the whole model and make sure it looks as expected
        ModelNode model = services.readWholeModel();
        //Useful for debugging :-)
        //System.out.println(model);
```

Now we make sure that the structure of the model within the controller has the expected format and values

```
Assert.assertTrue(model.get(SUBSYSTEM).hasDefined(SubsystemExtension.SUBSYSTEM_NAME));
        Assert.assertTrue(model.get(SUBSYSTEM,
SubsystemExtension.SUBSYSTEM_NAME).hasDefined("type"));
        Assert.assertTrue(model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME,
"type").hasDefined("tst"));
        Assert.assertTrue(model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME, "type",
"tst").hasDefined("tick"));
        Assert.assertEquals(12345, model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME,
"type", "tst", "tick").asLong());
    }
```

The last test provided is called `testParseAndMarshalModel()`. It's main purpose is to make sure that our `SubsystemParser.writeContent()` works as expected. This is achieved by starting a controller in the same way as before

```
@Test
    public void testParseAndMarshalModel() throws Exception {
        //Parse the subsystem xml and install into the first controller
        String subsystemXml =
                "<subsystem xmlns=\"" + SubsystemExtension.NAMESPACE + "\">" +
                "   <deployment-types>" +
                "       <deployment-type suffix=\"tst\" tick=\"12345\"/>" +
                "   </deployment-types>" +
                "</subsystem>";
        KernelServices servicesA = super.installInController(subsystemXml);
```

Now we read the model and the xml that was persisted from the first controller, and use that xml to start a second controller

```
//Get the model and the persisted xml from the first controller
        ModelNode modelA = servicesA.readWholeModel();
        String marshalled = servicesA.getPersistedSubsystemXml();

        //Install the persisted xml from the first controller into a second controller
        KernelServices servicesB = super.installInController(marshalled);
```

Finally we read the model from the second controller, and make sure that the models are identical by calling `compare()` on the test superclass.

```
ModelNode modelB = servicesB.readWholeModel();

        //Make sure the models from the two controllers are identical
        super.compare(modelA, modelB);
    }
```

We then have a test that needs no changing from what the archetype provides us with. As we have seen before we start a controller

```
@Test
    public void testDescribeHandler() throws Exception {
        //Parse the subsystem xml and install into the first controller
        String subsystemXml =
                "<subsystem xmlns=\"" + SubsystemExtension.NAMESPACE + "\">" +
                "</subsystem>";
        KernelServices servicesA = super.installInController(subsystemXml);
```

We then call `/subsystem=tracker:describe` which outputs the subsystem as operations needed to reach the current state (Done by our `SubsystemDescribeHandler`)

```
//Get the model and the describe operations from the first controller
        ModelNode modelA = servicesA.readWholeModel();
        ModelNode describeOp = new ModelNode();
        describeOp.get(OP).set(DESCRIBE);
        describeOp.get(OP_ADDR).set(
                PathAddress.pathAddress(
                        PathElement.pathElement(SUBSYSTEM,
SubsystemExtension.SUBSYSTEM_NAME)).toModelNode());
        List<ModelNode> operations =
super.checkResultAndGetContents(servicesA.executeOperation(describeOp)).asList();
```

Then we create a new controller using those operations

```
//Install the describe options from the first controller into a second controller
        KernelServices servicesB = super.installInController(operations);
```

And then we read the model from the second controller and make sure that the two subsystems are identical
ModelNode modelB = servicesB.readWholeModel();

```
//Make sure the models from the two controllers are identical
        super.compare(modelA, modelB);

    }
```

To test the removal of the the subsystem and child resources we modify the `testSubsystemRemoval()` test provided by the archetype:

```
/**
     * Tests that the subsystem can be removed
     */
    @Test
    public void testSubsystemRemoval() throws Exception {
        //Parse the subsystem xml and install into the first controller
```

We provide xml for the subsystem installing a child, which in turn installs a TrackerService

```
String subsystemXml =
                "<subsystem xmlns=\"" + SubsystemExtension.NAMESPACE + "\">" +
                "   <deployment-types>" +
                "       <deployment-type suffix=\"tst\" tick=\"12345\"/>" +
                "   </deployment-types>" +
                "</subsystem>";
        KernelServices services = super.installInController(subsystemXml);
```

Having installed the xml into the controller we make sure the TrackerService is there

```
//Sanity check to test the service for 'tst' was there
        services.getContainer().getRequiredService(TrackerService.createServiceName("tst"));
```

This call from the subsystem test harness will call remove for each level in our subsystem, children first and validate
that the subsystem model is empty at the end.

```
//Checks that the subsystem was removed from the model
        super.assertRemoveSubsystemResources(services);
```

Finally we check that all the services were removed by the remove handlers

```
//Check that any services that were installed were removed here
        try {
            services.getContainer().getRequiredService(TrackerService.createServiceName("tst"));
            Assert.fail("Should have removed services");
        } catch (Exception expected) {
        }
    }
```

For good measure let us throw in another test which adds a `deployment-type` and also changes its attribute at runtime. So first of all boot up the controller with the same xml we have been using so far

```
@Test
    public void testExecuteOperations() throws Exception {
        String subsystemXml =
                "<subsystem xmlns=\"" + SubsystemExtension.NAMESPACE + "\">" +
                "   <deployment-types>" +
                "       <deployment-type suffix=\"tst\" tick=\"12345\"/>" +
                "   </deployment-types>" +
                "</subsystem>";
        KernelServices services = super.installInController(subsystemXml);
```

Now create an operation which does the same as the following CLI command
`/subsystem=tracker/type=foo:add(tick=1000)`

```
//Add another type
        PathAddress fooTypeAddr = PathAddress.pathAddress(
                PathElement.pathElement(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME),
                PathElement.pathElement("type", "foo"));
        ModelNode addOp = new ModelNode();
        addOp.get(OP).set(ADD);
        addOp.get(OP_ADDR).set(fooTypeAddr.toModelNode());
        addOp.get("tick").set(1000);
```

Execute the operation and make sure it was successful

```
ModelNode result = services.executeOperation(addOp);
        Assert.assertEquals(SUCCESS, result.get(OUTCOME).asString());
```

Read the whole model and make sure that the original data is still there (i.e. the same as what was done by `testInstallIntoController()`

```
ModelNode model = services.readWholeModel();
        Assert.assertTrue(model.get(SUBSYSTEM).hasDefined(SubsystemExtension.SUBSYSTEM_NAME));
        Assert.assertTrue(model.get(SUBSYSTEM,
SubsystemExtension.SUBSYSTEM_NAME).hasDefined("type"));
        Assert.assertTrue(model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME,
"type").hasDefined("tst"));
        Assert.assertTrue(model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME, "type",
"tst").hasDefined("tick"));
        Assert.assertEquals(12345, model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME,
"type", "tst", "tick").asLong());
```

Then make sure our new `type` has been added:

```
Assert.assertTrue(model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME,
"type").hasDefined("foo"));
        Assert.assertTrue(model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME, "type",
"foo").hasDefined("tick"));
        Assert.assertEquals(1000, model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME,
"type", "foo", "tick").asLong());
```

Then we call `write-attribute` to change the `tick` value of `/subsystem=tracker/type=foo`:

```
//Call write-attribute
        ModelNode writeOp = new ModelNode();
        writeOp.get(OP).set(WRITE_ATTRIBUTE_OPERATION);
        writeOp.get(OP_ADDR).set(fooTypeAddr.toModelNode());
        writeOp.get(NAME).set("tick");
        writeOp.get(VALUE).set(3456);
        result = services.executeOperation(writeOp);
        Assert.assertEquals(SUCCESS, result.get(OUTCOME).asString());
```

To give you exposure to other ways of doing things, now instead of reading the whole model to check the attribute, we call `read-attribute` instead, and make sure it has the value we set it to.

```
//Check that write attribute took effect, this time by calling read-attribute instead of reading
the whole model
        ModelNode readOp = new ModelNode();
        readOp.get(OP).set(READ_ATTRIBUTE_OPERATION);
        readOp.get(OP_ADDR).set(fooTypeAddr.toModelNode());
        readOp.get(NAME).set("tick");
        result = services.executeOperation(readOp);
        Assert.assertEquals(3456, checkResultAndGetContents(result).asLong());
```

Since each `type` installs its own copy of `TrackerService`, we get the `TrackerService` for `type=foo` from the service container exposed by the kernel services and make sure it has the right value

```
TrackerService service =
(TrackerService)services.getContainer().getService(TrackerService.createServiceName("foo")).getVal
Assert.assertEquals(3456, service.getTick());
    }
```

TypeDefinition.TICK.

11.11.5 Add the deployers

When discussing `SubsystemAddHandler` we did not mention the work done to install the deployers, which is done in the following method:

```
@Override
    public void performBoottime(OperationContext context, ModelNode operation, ModelNode model,
            ServiceVerificationHandler verificationHandler, List<ServiceController<?>>
newControllers)
            throws OperationFailedException {

        log.info("Populating the model");

        //Add deployment processors here
        //Remove this if you don't need to hook into the deployers, or you can add as many as
you like
        //see SubDeploymentProcessor for explanation of the phases
        context.addStep(new AbstractDeploymentChainStep() {
            public void execute(DeploymentProcessorTarget processorTarget) {
                processorTarget.addDeploymentProcessor(SubsystemDeploymentProcessor.PHASE,
SubsystemDeploymentProcessor.priority, new SubsystemDeploymentProcessor());

            }
        }, OperationContext.Stage.RUNTIME);

    }
```

This adds an extra step which is responsible for installing deployment processors. You can add as many as you like, or avoid adding any all together depending on your needs. Each processor has a `Phase` and a `priority`. Phases are sequential, and a deployment passes through each phases deployment processors. The `priority` specifies where within a phase the processor appears. See `org.jboss.as.server.deployment.Phase` for more information about phases.

In our case we are keeping it simple and staying with one deployment processor with the phase and priority created for us by the maven archetype. The phases will be explained in the next section. The deployment processor is as follows:

```
public class SubsystemDeploymentProcessor implements DeploymentUnitProcessor {
    ...

    @Override
    public void deploy(DeploymentPhaseContext phaseContext) throws
DeploymentUnitProcessingException {
        String name = phaseContext.getDeploymentUnit().getName();
        TrackerService service = getTrackerService(phaseContext.getServiceRegistry(), name);
        if (service != null) {
            ResourceRoot root =
phaseContext.getDeploymentUnit().getAttachment(Attachments.DEPLOYMENT_ROOT);
            VirtualFile cool = root.getRoot().getChild("META-INF/cool.txt");
            service.addDeployment(name);
            if (cool.exists()) {
                service.addCoolDeployment(name);
            }
        }
    }

    @Override
    public void undeploy(DeploymentUnit context) {
        context.getServiceRegistry();
        String name = context.getName();
        TrackerService service = getTrackerService(context.getServiceRegistry(), name);
        if (service != null) {
            service.removeDeployment(name);
        }
    }

    private TrackerService getTrackerService(ServiceRegistry registry, String name) {
        int last = name.lastIndexOf(".");
        String suffix = name.substring(last + 1);
        ServiceController<?> container =
registry.getService(TrackerService.createServiceName(suffix));
        if (container != null) {
            TrackerService service = (TrackerService)container.getValue();
            return service;
        }
        return null;
    }
}
```

The `deploy()` method is called when a deployment is being deployed. In this case we look for the `TrackerService` instance for the service name created from the deployment's suffix. If there is one it means that we are meant to be tracking deployments with this suffix (i.e. `TypeAddHandler` was called for this suffix), and if we find one we add the deployment's name to it. Similarly `undeploy()` is called when a deployment is being undeployed, and if there is a `TrackerService` instance for the deployment's suffix, we remove the deployment's name from it.

Deployment phases and attachments

The code in the SubsystemDeploymentProcessor uses an *attachment*, which is the means of communication between the individual deployment processors. A deployment processor belonging to a phase may create an attachment which is then read further along the chain of deployment unit processors. In the above example we look for the `Attachments.DEPLOYMENT_ROOT` attachment, which is a view of the file structure of the deployment unit put in place before the chain of deployment unit processors is invoked.

As mentioned above, the deployment unit processors are organized in phases, and have a relative order within each phase. A deployment unit passes through all the deployment unit processors in that order. A deployment unit processor may choose to take action or not depending on what attachments are available. Let's take a quick look at what the deployment unit processors for in the phases described in `org.jboss.as.server.deployment.Phase`.

STRUCTURE

The deployment unit processors in this phase determine the structure of a deployment, and looks for sub deployments and metadata files.

PARSE

In this phase the deployment unit processors parse the deployment descriptors and build up the annotation index. `Class-Path` entries from the META-INF/MANIFEST.MF are added.

DEPENDENCIES

Extra class path dependencies are added. For example if deploying a `war` file, the commonly needed dependencies for a web application are added.

CONFIGURE_MODULE

In this phase the modular class loader for the deployment is created. No attempt should be made loading classes from the deployment until **after** this phase.

POST_MODULE

Now that our class loader has been constructed we have access to the classes. In this stage deployment processors may use the `Attachments.REFLECTION_INDEX` attachment which is a deployment index used to obtain members of classes in the deployment, and to invoke upon them, bypassing the inefficiencies of using `java.lang.reflect` directly.

INSTALL

Install new services coming from the deployment.

CLEANUP

Attachments put in place earlier in the deployment unit processor chain may be removed here.

11.11.6 Integrate with WildFly

Now that we have all the code needed for our subsystem, we can build our project by running `mvn install`

```
[kabir ~/sourcecontrol/temp/archetype-test/acme-subsystem]
$mvn install
[INFO] Scanning for projects...
[...]
main:
   [delete] Deleting:
/Users/kabir/sourcecontrol/temp/archetype-test/acme-subsystem/null1004283288
   [delete] Deleting directory
/Users/kabir/sourcecontrol/temp/archetype-test/acme-subsystem/target/module
     [copy] Copying 1 file to
/Users/kabir/sourcecontrol/temp/archetype-test/acme-subsystem/target/module/com/acme/corp/tracker/
[copy] Copying 1 file to
/Users/kabir/sourcecontrol/temp/archetype-test/acme-subsystem/target/module/com/acme/corp/tracker/
[echo] Module com.acme.corp.tracker has been created in the target/module directory. Copy to
your JBoss AS 7 installation.
[INFO] Executed tasks
[INFO]
[INFO] --- maven-install-plugin:2.3.1:install (default-install) @ acme-subsystem ---
[INFO] Installing
/Users/kabir/sourcecontrol/temp/archetype-test/acme-subsystem/target/acme-subsystem.jar to
/Users/kabir/.m2/repository/com/acme/corp/acme-subsystem/1.0-SNAPSHOT/acme-subsystem-1.0-SNAPSHOT.
Installing /Users/kabir/sourcecontrol/temp/archetype-test/acme-subsystem/pom.xml to
/Users/kabir/.m2/repository/com/acme/corp/acme-subsystem/1.0-SNAPSHOT/acme-subsystem-1.0-SNAPSHOT.
------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 5.851s
[INFO] Finished at: Mon Jul 11 23:24:58 BST 2011
[INFO] Final Memory: 7M/81M
[INFO] ------------------------------------------------------------------------
```

This will have built our project and assembled a module for us that can be used for installing it into WildFly. If you go to the `target/module` folder where you built the project you will see the module

```
$ls target/module/com/acme/corp/tracker/main/
acme-subsystem.jar  module.xml
```

The `module.xml` comes from `src/main/resources/module/main/module.xml` and is used to define your module. It says that it contains the `acme-subsystem.jar`:

```
<module xmlns="urn:jboss:module:1.0" name="com.acme.corp.tracker">
    <resources>
        <resource-root path="acme-subsystem.jar"/>
    </resources>
```

And has a default set of dependencies needed by every subsystem created. If your subsystem requires additional module dependencies you can add them here before building and installing.

```
<dependencies>
        <module name="javax.api"/>
        <module name="org.jboss.staxmapper"/>
        <module name="org.jboss.as.controller"/>
        <module name="org.jboss.as.server"/>
        <module name="org.jboss.modules"/>
        <module name="org.jboss.msc"/>
        <module name="org.jboss.logging"/>
        <module name="org.jboss.vfs"/>
    </dependencies>
</module>
```

Note that the name of the module corresponds to the directory structure containing it. Now copy the `target/module/com/acme/corp/tracker/main/` directory and its contents to `$WFLY/modules/com/acme/corp/tracker/main/` (where `$WFLY` is the root of your WildFly install).

Next we need to modify `$WFLY/standalone/configuration/standalone.xml`. First we need to add our new module to the `<extensions>` section:

```
<extensions>
        ...
        <extension module="org.jboss.as.weld"/>
        <extension module="com.acme.corp.tracker"/>
    </extensions>
```

And then we have to add our subsystem to the `<profile>` section:

```
<profile>
    ...

        <subsystem xmlns="urn:com.acme.corp.tracker:1.0">
            <deployment-types>
                <deployment-type suffix="sar" tick="10000"/>
                <deployment-type suffix="war" tick="10000"/>
            </deployment-types>
        </subsystem>
    ...
    </profile>
```

Adding this to a managed domain works exactly the same apart from in this case you need to modify `$WFLY/domain/configuration/domain.xml`.

Now start up WildFly by running `$WFLY/bin/standalone.sh` and you should see messages like these after the server has started, which means our subsystem has been added and our `TrackerService` is working:

```
15:27:33,838 INFO  [org.jboss.as] (Controller Boot Thread) JBoss AS 7.0.0.Final "Lightning"
started in 2861ms - Started 94 of 149 services (55 services are passive or on-demand)
15:27:42,966 INFO  [stdout] (Thread-8) Current deployments deployed while sar tracking active:
15:27:42,966 INFO  [stdout] (Thread-8) []
15:27:42,967 INFO  [stdout] (Thread-8) Cool: 0
15:27:42,967 INFO  [stdout] (Thread-9) Current deployments deployed while war tracking active:
15:27:42,967 INFO  [stdout] (Thread-9) []
15:27:42,967 INFO  [stdout] (Thread-9) Cool: 0
15:27:52,967 INFO  [stdout] (Thread-8) Current deployments deployed while sar tracking active:
15:27:52,967 INFO  [stdout] (Thread-8) []
15:27:52,967 INFO  [stdout] (Thread-8) Cool: 0
```

If you run the command line interface you can execute some commands to see more about the subsystem. For example

```
[standalone@localhost:9999 /] /subsystem=tracker/:read-resource-description(recursive=true,
operations=true)
```

will return a lot of information, including what we provided in the `DescriptionProviders` we created to document our subsystem.

To see the current subsystem state you can execute

```
[standalone@localhost:9999 /] /subsystem=tracker/:read-resource(recursive=true)
{
    "outcome" => "success",
    "result" => {"type" => {
        "war" => {"tick" => 10000L},
        "sar" => {"tick" => 10000L}
    }}
}
```

We can remove both the deployment types which removes them from the model:

```
[standalone@localhost:9999 /] /subsystem=tracker/type=sar:remove
{"outcome" => "success"}
[standalone@localhost:9999 /] /subsystem=tracker/type=war:remove
{"outcome" => "success"}
[standalone@localhost:9999 /] /subsystem=tracker/:read-resource(recursive=true)
{
    "outcome" => "success",
    "result" => {"type" => undefined}
}
```

You should now see the output from the `TrackerService` instances having stopped.

Now, let's add the war tracker again:

```
[standalone@localhost:9999 /] /subsystem=tracker/type=war:add
{"outcome" => "success"}
[standalone@localhost:9999 /] /subsystem=tracker/:read-resource(recursive=true)
{
    "outcome" => "success",
    "result" => {"type" => {"war" => {"tick" => 10000L}}}
}
```

and the WildFly console should show the messages coming from the war `TrackerService` again.

Now let us deploy something. You can find two maven projects for test wars already built at test1.zip and test2.zip. If you download them and extract them to `/Downloads/test1` and `/Downloads/test2`, you can see that `/Downloads/test1/target/test1.war` contains a `META-INF/cool.txt` while `/Downloads/test2/target/test2.war` does not contain that file. From CLI deploy `test1.war` first:

```
[standalone@localhost:9999 /] deploy ~/Downloads/test1/target/test1.war
'test1.war' deployed successfully.
```

And you should now see the output from the war `TrackerService` list the deployments:

```
15:35:03,712 INFO  [org.jboss.as.server.deployment] (MSC service thread 1-2) Starting deployment
of "test1.war"
15:35:03,988 INFO  [org.jboss.web] (MSC service thread 1-1) registering web context: /test1
15:35:03,996 INFO  [org.jboss.as.server.controller] (pool-2-thread-9) Deployed "test1.war"
15:35:13,056 INFO  [stdout] (Thread-9) Current deployments deployed while war tracking active:
15:35:13,056 INFO  [stdout] (Thread-9) [test1.war]
15:35:13,057 INFO  [stdout] (Thread-9) Cool: 1
```

So our `test1.war` got picked up as a 'cool' deployment. Now if we deploy `test2.war`

```
[standalone@localhost:9999 /] deploy ~/sourcecontrol/temp/archetype-test/test2/target/test2.war
'test2.war' deployed successfully.
```

You will see that deployment get picked up as well but since there is no `META-INF/cool.txt` it is not marked as a 'cool' deployment:

```
15:37:05,634 INFO  [org.jboss.as.server.deployment] (MSC service thread 1-4) Starting deployment
of "test2.war"
15:37:05,699 INFO  [org.jboss.web] (MSC service thread 1-1) registering web context: /test2
15:37:05,982 INFO  [org.jboss.as.server.controller] (pool-2-thread-15) Deployed "test2.war"
15:37:13,075 INFO  [stdout] (Thread-9) Current deployments deployed while war tracking active:
15:37:13,075 INFO  [stdout] (Thread-9) [test1.war, test2.war]
15:37:13,076 INFO  [stdout] (Thread-9) Cool: 1
```

An undeploy

```
[standalone@localhost:9999 /] undeploy test1.war
Successfully undeployed test1.war.
```

is also reflected in the `TrackerService` output:

```
15:38:47,901 INFO  [org.jboss.as.server.controller] (pool-2-thread-21) Undeployed "test1.war"
15:38:47,934 INFO  [org.jboss.as.server.deployment] (MSC service thread 1-3) Stopped deployment
test1.war in 40ms
15:38:53,091 INFO  [stdout] (Thread-9) Current deployments deployed while war tracking active:
15:38:53,092 INFO  [stdout] (Thread-9) [test2.war]
15:38:53,092 INFO  [stdout] (Thread-9) Cool: 0
```

Finally, we registered a write attribute handler for the `tick` property of the `type` so we can change the frequency

```
[standalone@localhost:9999 /] /subsystem=tracker/type=war:write-attribute(name=tick,value=1000)
{"outcome" => "success"}
```

You should now see the output from the `TrackerService` happen every second

```
15:39:43,100 INFO  [stdout] (Thread-9) Current deployments deployed while war tracking active:
15:39:43,100 INFO  [stdout] (Thread-9) [test2.war]
15:39:43,101 INFO  [stdout] (Thread-9) Cool: 0
15:39:44,101 INFO  [stdout] (Thread-9) Current deployments deployed while war tracking active:
15:39:44,102 INFO  [stdout] (Thread-9) [test2.war]
15:39:44,105 INFO  [stdout] (Thread-9) Cool: 0
15:39:45,106 INFO  [stdout] (Thread-9) Current deployments deployed while war tracking active:
15:39:45,106 INFO  [stdout] (Thread-9) [test2.war]
```

If you open `$WFLY/standalone/configuration/standalone.xml` you can see that our subsystem entry reflects the current state of the subsystem:

```
<subsystem xmlns="urn:com.acme.corp.tracker:1.0">
            <deployment-types>
                <deployment-type suffix="war" tick="1000"/>
            </deployment-types>
        </subsystem>
```

11.11.7 Expressions

Expressions are mechanism that enables you to support variables in your attributes, for instance when you want the value of attribute to be resolved using system / environment properties.

An example expression is

```
${jboss.bind.address.management:127.0.0.1}
```

which means that the value should be taken from a system property named `jboss.bind.address.management` and if it is not defined use `127.0.0.1`.

What expression types are supported

- System properties, which are resolved using `java.lang.System.getProperty(String key)`
- Environment properties, which are resolved using `java.lang.System.getEnv(String name)`.
- Security vault expressions, resolved against the security vault configured for the server or Host Controller that needs to resolve the expression.

In all cases, the syntax for the expression is

```
${expression_to_resolve}
```

For an expression meant to be resolved against environment properties, the `expression_to_resolve` must be prefixed with `env.`. The portion after `env.` will be the name passed to `java.lang.System.getEnv(String name)`.

Security vault expressions do not support default values (i.e. the `127.0.0.1` in the `jboss.bind.address.management:127.0.0.1` example above.)

How to support expressions in subsystems

The easiest way is by using AttributeDefinition, which provides support for expressions just by using it correctly.

When we create an AttributeDefinition all we need to do is mark that is allows expressions. Here is an example how to define an attribute that allows expressions to be used.

```
SimpleAttributeDefinition MY_ATTRIBUTE =
            new SimpleAttributeDefinitionBuilder("my-attribute", ModelType.INT, true)
                    .setAllowExpression(true)
                    .setFlags(AttributeAccess.Flag.RESTART_ALL_SERVICES)
                    .setDefaultValue(new ModelNode(1))
                    .build();
```

Then later when you are parsing the xml configuration you should use the MY_ATTRIBUTE attribute definition to set the value to the management operation ModelNode you are creating.

```
....
      String attr = reader.getAttributeLocalName(i);
      String value = reader.getAttributeValue(i);
      if (attr.equals("my-attribute")) {
          MY_ATTRIBUTE.parseAndSetParameter(value, operation, reader);
      } else if (attr.equals("suffix")) {
.....
```

Note that this just helps you to properly set the value to the model node you are working on, so no need to additionally set anything to the model for this attribute. Method parseAndSetParameter parses the value that was read from xml for possible expressions in it and if it finds any it creates special model node that defines that node is of type `ModelType.EXPRESSION`.

Later in your operation handlers where you implement populateModel and have to store the value from the operation to the configuration model you also use this MY_ATTRIBUTE attribute definition.

```
@Override
 protected void populateModel(ModelNode operation, ModelNode model) throws
OperationFailedException {
        MY_ATTRIBUTE.validateAndSet(operation,model);
 }
```

This will make sure that the attribute that is stored from the operation to the model is valid and nothing is lost. It also checks the value stored in the operation `ModelNode`, and if it isn't already `ModelType.EXPRESSION`, it checks if the value is a string that contains the expression syntax. If so, the value stored in the model will be of type `ModelType.EXPRESSION`. Doing this ensures that expressions are properly handled when they appear in operations that weren't created by the subsystem parser, but are instead passed in from CLI or admin console users.

As last step we need to use the value of the attribute. This is usually needed inside of the `performRuntime` method

```
protected void performRuntime(OperationContext context, ModelNode operation, ModelNode model,
ServiceVerificationHandler verificationHandler, List<ServiceController<?>> newControllers)
throws OperationFailedException {
       ....
        final int attributeValue = MY_ATTRIBUTE.resolveModelAttribute(context,
model).asInt();
        ...

    }
```

As you can see resolving of attribute's value is not done until it is needed for use in the subsystem's runtime services. The resolved value is not stored in the configuration model, the unresolved expression is. That way we do not lose any information in the model and can assure that also marshalling is done properly, where we must marshall back the unresolved value.

Attribute definitinon also helps you with that:

```
public void writeContent(XMLExtendedStreamWriter writer, SubsystemMarshallingContext context)
throws XMLStreamException {
    ....
      MY_ATTRIBUTE.marshallAsAttribute(sessionData, writer);
      MY_OTHER_ATTRIBUTE.marshallAsElement(sessionData, false, writer);
    ...
}
```

11.11.8 Add the deployers

When discussing `SubsystemAddHandler` we did not mention the work done to install the deployers, which is done in the following method:

```
@Override
    public void performBoottime(OperationContext context, ModelNode operation, ModelNode model,
            ServiceVerificationHandler verificationHandler, List<ServiceController<?>>
newControllers)
            throws OperationFailedException {

        log.info("Populating the model");

        //Add deployment processors here
        //Remove this if you don't need to hook into the deployers, or you can add as many as
you like
        //see SubDeploymentProcessor for explanation of the phases
        context.addStep(new AbstractDeploymentChainStep() {
            public void execute(DeploymentProcessorTarget processorTarget) {
                processorTarget.addDeploymentProcessor(SubsystemDeploymentProcessor.PHASE,
SubsystemDeploymentProcessor.priority, new SubsystemDeploymentProcessor());

            }
        }, OperationContext.Stage.RUNTIME);

    }
```

This adds an extra step which is responsible for installing deployment processors. You can add as many as you like, or avoid adding any all together depending on your needs. Each processor has a `Phase` and a `priority`. Phases are sequential, and a deployment passes through each phases deployment processors. The `priority` specifies where within a phase the processor appears. See `org.jboss.as.server.deployment.Phase` for more information about phases.

In our case we are keeping it simple and staying with one deployment processor with the phase and priority created for us by the maven archetype. The phases will be explained in the next section. The deployment processor is as follows:

```
public class SubsystemDeploymentProcessor implements DeploymentUnitProcessor {
    ...

    @Override
    public void deploy(DeploymentPhaseContext phaseContext) throws
DeploymentUnitProcessingException {
        String name = phaseContext.getDeploymentUnit().getName();
        TrackerService service = getTrackerService(phaseContext.getServiceRegistry(), name);
        if (service != null) {
            ResourceRoot root =
phaseContext.getDeploymentUnit().getAttachment(Attachments.DEPLOYMENT_ROOT);
            VirtualFile cool = root.getRoot().getChild("META-INF/cool.txt");
            service.addDeployment(name);
            if (cool.exists()) {
                service.addCoolDeployment(name);
            }
        }
    }

    @Override
    public void undeploy(DeploymentUnit context) {
        context.getServiceRegistry();
        String name = context.getName();
        TrackerService service = getTrackerService(context.getServiceRegistry(), name);
        if (service != null) {
            service.removeDeployment(name);
        }
    }

    private TrackerService getTrackerService(ServiceRegistry registry, String name) {
        int last = name.lastIndexOf(".");
        String suffix = name.substring(last + 1);
        ServiceController<?> container =
registry.getService(TrackerService.createServiceName(suffix));
        if (container != null) {
            TrackerService service = (TrackerService)container.getValue();
            return service;
        }
        return null;
    }
}
```

The `deploy()` method is called when a deployment is being deployed. In this case we look for the `TrackerService` instance for the service name created from the deployment's suffix. If there is one it means that we are meant to be tracking deployments with this suffix (i.e. `TypeAddHandler` was called for this suffix), and if we find one we add the deployment's name to it. Similarly `undeploy()` is called when a deployment is being undeployed, and if there is a `TrackerService` instance for the deployment's suffix, we remove the deployment's name from it.

Deployment phases and attachments

The code in the SubsystemDeploymentProcessor uses an *attachment*, which is the means of communication between the individual deployment processors. A deployment processor belonging to a phase may create an attachment which is then read further along the chain of deployment unit processors. In the above example we look for the `Attachments.DEPLOYMENT_ROOT` attachment, which is a view of the file structure of the deployment unit put in place before the chain of deployment unit processors is invoked.

As mentioned above, the deployment unit processors are organized in phases, and have a relative order within each phase. A deployment unit passes through all the deployment unit processors in that order. A deployment unit processor may choose to take action or not depending on what attachments are available. Let's take a quick look at what the deployment unit processors for in the phases described in `org.jboss.as.server.deployment.Phase`.

STRUCTURE

The deployment unit processors in this phase determine the structure of a deployment, and looks for sub deployments and metadata files.

PARSE

In this phase the deployment unit processors parse the deployment descriptors and build up the annotation index. `Class-Path` entries from the META-INF/MANIFEST.MF are added.

DEPENDENCIES

Extra class path dependencies are added. For example if deploying a `war` file, the commonly needed dependencies for a web application are added.

CONFIGURE_MODULE

In this phase the modular class loader for the deployment is created. No attempt should be made loading classes from the deployment until **after** this phase.

POST_MODULE

Now that our class loader has been constructed we have access to the classes. In this stage deployment processors may use the `Attachments.REFLECTION_INDEX` attachment which is a deployment index used to obtain members of classes in the deployment, and to invoke upon them, bypassing the inefficiencies of using `java.lang.reflect` directly.

INSTALL

Install new services coming from the deployment.

CLEANUP

Attachments put in place earlier in the deployment unit processor chain may be removed here.

11.11.9 Create the schema

First, let us define the schema for our subsystem. Rename `src/main/resources/schema/mysubsystem.xsd` to `src/main/resources/schema/acme.xsd`. Then open `acme.xsd` and modify it to the following

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
            targetNamespace="urn:com.acme.corp.tracker:1.0"
            xmlns="urn:com.acme.corp.tracker:1.0"
            elementFormDefault="qualified"
            attributeFormDefault="unqualified"
            version="1.0">

   <!-- The subsystem root element -->
   <xs:element name="subsystem" type="subsystemType"/>
   <xs:complexType name="subsystemType">
      <xs:all>
         <xs:element name="deployment-types" type="deployment-typesType"/>
      </xs:all>
   </xs:complexType>
   <xs:complexType name="deployment-typesType">
      <xs:choice minOccurs="0" maxOccurs="unbounded">
         <xs:element name="deployment-type" type="deployment-typeType"/>
      </xs:choice>
   </xs:complexType>
   <xs:complexType name="deployment-typeType">
      <xs:attribute name="suffix" use="required"/>
      <xs:attribute name="tick" type="xs:long" use="optional" default="10000"/>
   </xs:complexType>
</xs:schema>
```

Note that we modified the `xmlns` and `targetNamespace` values to `urn.com.acme.corp.tracker:1.0`. Our new `subsystem` element has a child called `deployment-types`, which in turn can have zero or more children called `deployment-type`. Each `deployment-type` has a required `suffix` attribute, and a `tick` attribute which defaults to `true`.
Now modify the `com.acme.corp.tracker.extension.SubsystemExtension` class to contain the new namespace.

```
public class SubsystemExtension implements Extension {

    /** The name space used for the {@code substystem} element */
    public static final String NAMESPACE = "urn:com.acme.corp.tracker:1.0";
    ...
```

11.11.10 Create the skeleton project

To make your life easier we have provided a maven archetype which will create a skeleton project for implementing subsystems.

```
mvn archetype:generate \
    -DarchetypeArtifactId=wildfly-subsystem \
    -DarchetypeGroupId=org.wildfly.archetypes \
    -DarchetypeVersion=8.0.0.Final \
    -DarchetypeRepository=http://repository.jboss.org/nexus/content/groups/public
```

Maven will download the archetype and it's dependencies, and ask you some questions:

```
$ mvn archetype:generate \
    -DarchetypeArtifactId=wildfly-subsystem \
    -DarchetypeGroupId=org.wildfly.archetypes \
    -DarchetypeVersion=8.0.0.Final \
    -DarchetypeRepository=http://repository.jboss.org/nexus/content/groups/public
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO] ------------------------------------------------------------------------
[INFO]

.........

Define value for property 'groupId': : com.acme.corp
Define value for property 'artifactId': : acme-subsystem
Define value for property 'version':  1.0-SNAPSHOT: :
Define value for property 'package':  com.acme.corp: : com.acme.corp.tracker
Define value for property 'module': : com.acme.corp.tracker
[INFO] Using property: name = WildFly subsystem project
Confirm properties configuration:
groupId: com.acme.corp
artifactId: acme-subsystem
version: 1.0-SNAPSHOT
package: com.acme.corp.tracker
module: com.acme.corp.tracker
name: WildFly subsystem project
 Y: : Y
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1:42.563s
[INFO] Finished at: Fri Jul 08 14:30:09 BST 2011
[INFO] Final Memory: 7M/81M
[INFO] ------------------------------------------------------------------------
$
```

	Instruction
1	Enter the `groupId` you wish to use
2	Enter the `artifactId` you wish to use
3	Enter the version you wish to use, or just hit `Enter` if you wish to accept the default `1.0-SNAPSHOT`
4	Enter the java package you wish to use, or just hit `Enter` if you wish to accept the default (which is copied from `groupId`).
5	Enter the module name you wish to use for your extension.
6	Finally, if you are happy with your choices, hit `Enter` and Maven will generate the project for you.

You can also do this in Eclipse, see Creating your own application for more details. We now have a skeleton project that you can use to implement a subsystem. Import the `acme-subsystem` project into your favourite IDE. A nice side-effect of running this in the IDE is that you can see the javadoc of WildFly classes and interfaces imported by the skeleton code. If you do a `mvn install` in the project it will work if we plug it into WildFly, but before doing that we will change it to do something more useful.

The rest of this section modifies the skeleton project created by the archetype to do something more useful, and the full code can be found in acme-subsystem.zip.

If you do a `mvn install` in the created project, you will see some tests being run

```
$mvn install
[INFO] Scanning for projects...
[...]
[INFO] Surefire report directory:
/Users/kabir/sourcecontrol/temp/archetype-test/acme-subsystem/target/surefire-reports

-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running com.acme.corp.tracker.extension.SubsystemBaseParsingTestCase
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.424 sec
Running com.acme.corp.tracker.extension.SubsystemParsingTestCase
Tests run: 6, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.074 sec

Results :

Tests run: 3, Failures: 0, Errors: 0, Skipped: 0
[...]
```

We will talk about these later in the Testing the parsers section.

11.11.11 Design and define the model structure

The following example xml contains a valid subsystem configuration, we will see how to plug this in to WildFly later in this tutorial.

```
<subsystem xmlns="urn:com.acme.corp.tracker:1.0">
   <deployment-types>
      <deployment-type suffix="sar" tick="10000"/>
      <deployment-type suffix="war" tick="10000"/>
   </deployment-types>
</subsystem>
```

Now when designing our model, we can either do a one to one mapping between the schema and the model or come up with something slightly or very different. To keep things simple, let us stay pretty true to the schema so that when executing a `:read-resource(recursive=true)` against our subsystem we'll see something like:

```
{
    "outcome" => "success",
    "result" => {"type" => {
        "sar" => {"tick" => "10000"},
        "war" => {"tick" => "10000"}
    }}
}
```

Each `deployment-type` in the xml becomes in the model a child resource of the subsystem's root resource. The child resource's child-type is `type`, and it is indexed by its `suffix`. Each `type` resource then contains the `tick` attribute.
We also need a name for our subsystem, to do that change `com.acme.corp.tracker.extension.SubsystemExtension`:

```
public class SubsystemExtension implements Extension {
    ...
    /** The name of our subsystem within the model. */
    public static final String SUBSYSTEM_NAME = "tracker";
    ...
```

Once we are finished our subsystem will be available under `/subsystem=tracker`.
The `SubsystemExtension.initialize()` method defines the model, currently it sets up the basics to add our subsystem to the model:

```
@Override
    public void initialize(ExtensionContext context) {
        //register subsystem with its model version
        final SubsystemRegistration subsystem = context.registerSubsystem(SUBSYSTEM_NAME, 1, 0);
        //register subsystem model with subsystem definition that defines all attributes and
operations
        final ManagementResourceRegistration registration =
subsystem.registerSubsystemModel(SubsystemDefinition.INSTANCE);
        //register describe operation, note that this can be also registered in
SubsystemDefinition
        registration.registerOperationHandler(DESCRIBE,
GenericSubsystemDescribeHandler.INSTANCE, GenericSubsystemDescribeHandler.INSTANCE, false,
OperationEntry.EntryType.PRIVATE);
        //we can register additional submodels here
        //
        subsystem.registerXMLElementWriter(parser);
    }
```

The `registerSubsystem()` call registers our subsystem with the extension context. At the end of the method we register our parser with the returned `SubsystemRegistration` to be able to marshal our subsystem's model back to the main configuration file when it is modified. We will add more functionality to this method later.

Registering the core subsystem model

Next we obtain a `ManagementResourceRegistration` by registering the subsystem model. This is a **compulsory** step for every new subsystem.

```
final ManagementResourceRegistration registration =
subsystem.registerSubsystemModel(SubsystemDefinition.INSTANCE);
```

Its parameter is an implementation of the `ResourceDefinition` interface, which means that when you call `/subsystem=tracker:read-resource-description` the information you see comes from model that is defined by `SubsystemDefinition.INSTANCE`.

```
public class SubsystemDefinition extends SimpleResourceDefinition {
    public static final SubsystemDefinition INSTANCE = new SubsystemDefinition();

    private SubsystemDefinition() {
        super(SubsystemExtension.SUBSYSTEM_PATH,
                SubsystemExtension.getResourceDescriptionResolver(null),
                //We always need to add an 'add' operation
                SubsystemAdd.INSTANCE,
                //Every resource that is added, normally needs a remove operation
                SubsystemRemove.INSTANCE);
    }

    @Override
    public void registerOperations(ManagementResourceRegistration resourceRegistration) {
        super.registerOperations(resourceRegistration);
        //you can register aditional operations here
    }

    @Override
    public void registerAttributes(ManagementResourceRegistration resourceRegistration) {
        //you can register attributes here
    }
}
```

Since we need child resource `type` we need to add new ResourceDefinition,

The `ManagementResourceRegistration` obtained in `SubsystemExtension.initialize()` is then used to add additional operations or to register submodels to the `/subsystem=tracker` address. Every subsystem and resource **must** have an `ADD` method which can be achieved by the following line inside `registerOperations` in your `ResourceDefinition` or by providing it in constructor of your `SimpleResourceDefinition` just as we did in example above.

```
//We always need to add an 'add' operation
        resourceRegistration.registerOperationHandler(ADD, SubsystemAdd.INSTANCE, new
DefaultResourceAddDescriptionProvider(resourceRegistration,descriptionResolver), false);
```

The parameters when registering an operation handler are:

1. **The name** - i.e. `ADD`.
2. The handler instance - we will talk more about this below
3. The handler description provider - we will talk more about this below.
4. Whether this operation handler is inherited - `false` means that this operation is not inherited, and will only apply to `/subsystem=tracker`. The content for this operation handler will be provided by 3.

Let us first look at the description provider which is quite simple since this operation takes no parameters. The addition of `type` children will be handled by another operation handler, as we will see later on.

There are two way to define `DescriptionProvider`, one is by defining it by hand using ModelNode, but as this has show to be very error prone there are lots of helper methods to help you automatically describe the model. Following example is done by manually defining Description provider for ADD operation handler

```
/**
     * Used to create the description of the subsystem add method
     */
    public static DescriptionProvider SUBSYSTEM_ADD = new DescriptionProvider() {
        public ModelNode getModelDescription(Locale locale) {
            //The locale is passed in so you can internationalize the strings used in the
descriptions

            final ModelNode subsystem = new ModelNode();
            subsystem.get(OPERATION_NAME).set(ADD);
            subsystem.get(DESCRIPTION).set("Adds the tracker subsystem");

            return subsystem;
        }
    };
```

Or you can use API that helps you do that for you. For Add and Remove methods there are classes `DefaultResourceAddDescriptionProvider` and `DefaultResourceRemoveDescriptionProvider` that do work for you. In case you use `SimpleResourceDefinition` even that part is hidden from you.

```
resourceRegistration.registerOperationHandler(ADD, SubsystemAdd.INSTANCE, new
DefaultResourceAddDescriptionProvider(resourceRegistration,descriptionResolver), false);
resourceRegistration.registerOperationHandler(REMOVE, SubsystemRemove.INSTANCE, new
DefaultResourceRemoveDescriptionProvider(resourceRegistration,descriptionResolver), false);
```

For other operation handlers that are not add/remove you can use `DefaultOperationDescriptionProvider` that takes additional parameter of what is the name of operation and optional array of parameters/attributes operation takes. This is an example to register operation "`add-mime`" with two parameters:

```
container.registerOperationHandler("add-mime",
                MimeMappingAdd.INSTANCE,
                new DefaultOperationDescriptionProvider("add-mime",
Extension.getResourceDescriptionResolver("container.mime-mapping"), MIME_NAME, MIME_VALUE));
```

When descriping an operation its description provider's `OPERATION_NAME` must match the name used when calling `ManagementResourceRegistration.registerOperationHandler()`

Next we have the actual operation handler instance, note that we have changed its `populateModel()` method to initialize the `type` child of the model.

```
class SubsystemAdd extends AbstractBoottimeAddStepHandler {

    static final SubsystemAdd INSTANCE = new SubsystemAdd();

    private SubsystemAdd() {
    }

    /** {@inheritDoc} */
    @Override
    protected void populateModel(ModelNode operation, ModelNode model) throws
OperationFailedException {
        log.info("Populating the model");
        //Initialize the 'type' child node
        model.get("type").setEmptyObject();
    }
    ....
```

`SubsystemAdd` also has a `performBoottime()` method which is used for initializing the deployer chain associated with this subsystem. We will talk about the deployers later on. However, the basic idea for all operation handlers is that we do any model updates before changing the actual runtime state.

The rule of thumb is that every thing that can be added, can also be removed so we have a remove handler for the subsystem registered
in `SubsystemDefinition.registerOperations` or just provide the operation handler in constructor.

```
//Every resource that is added, normally needs a remove operation
        registration.registerOperationHandler(REMOVE, SubsystemRemove.INSTANCE,
DefaultResourceRemoveDescriptionProvider(resourceRegistration,descriptionResolver) , false);
```

`SubsystemRemove` extends `AbstractRemoveStepHandler` which takes care of removing the resource from the model so we don't need to override its `performRemove()` operation, also the add handler did not install any services (services will be discussed later) so we can delete the `performRuntime()` method generated by the archetype.

```
class SubsystemRemove extends AbstractRemoveStepHandler {

    static final SubsystemRemove INSTANCE = new SubsystemRemove();

    private final Logger log = Logger.getLogger(SubsystemRemove.class);

    private SubsystemRemove() {
    }
}
```

The description provider for the remove operation is simple and quite similar to that of the add handler where just name of the method changes.

Registering the subsystem child

The `type` child does not exist in our skeleton project so we need to implement the operations to add and remove them from the model.

First we need an add operation to add the `type` child, create a class called `com.acme.corp.tracker.extension.TypeAddHandler`. In this case we extend the `org.jboss.as.controller.AbstractAddStepHandler` class and implement the `org.jboss.as.controller.descriptions.DescriptionProvider` interface. `org.jboss.as.controller.OperationStepHandler` is the main interface for the operation handlers, and `AbstractAddStepHandler` is an implementation of that which does the plumbing work for adding a resource to the model.

```
class TypeAddHandler extends AbstractAddStepHandler implements DescriptionProvider {

    public static final TypeAddHandler INSTANCE = new TypeAddHandler();

    private TypeAddHandler() {
    }
```

Then we define subsystem model. Lets call it `TypeDefinition` and for ease of use let it extend `SimpleResourceDefinition` instead just implement `ResourceDefinition`.

```
public class TypeDefinition extends SimpleResourceDefinition {

 public static final TypeDefinition INSTANCE = new TypeDefinition();

 //we define attribute named tick
protected static final SimpleAttributeDefinition TICK =
new SimpleAttributeDefinitionBuilder(TrackerExtension.TICK, ModelType.LONG)
  .setAllowExpression(true)
  .setXmlName(TrackerExtension.TICK)
  .setFlags(AttributeAccess.Flag.RESTART_ALL_SERVICES)
  .setDefaultValue(new ModelNode(1000))
  .setAllowNull(false)
  .build();

private TypeDefinition(){
   super(TYPE_PATH,
TrackerExtension.getResourceDescriptionResolver(TYPE),TypeAdd.INSTANCE,TypeRemove.INSTANCE);
}

@Override
public void registerAttributes(ManagementResourceRegistration resourceRegistration){
   resourceRegistration.registerReadWriteAttribute(TICK, null, TrackerTickHandler.INSTANCE);
}

}
```

Which will take care of describing the model for us. As you can see in example above we define `SimpleAttributeDefinition` named `TICK`, this is a mechanism to define Attributes in more type safe way and to add more common API to manipulate attributes. As you can see here we define default value of 1000 as also other constraints and capabilities. There could be other properties set such as validators, alternate names, xml name, flags for marking it attribute allows expressions and more.
Then we do the work of updating the model by implementing the `populateModel()` method from the `AbstractAddStepHandler`, which populates the model's attribute from the operation parameters. First we get hold of the model relative to the address of this operation (we will see later that we will register it against `/subsystem=tracker/type=*`), so we just specify an empty relative address, and we then populate our model with the parameters from the operation. There is operation `validateAndSet` on `AttributeDefinition` that helps us validate and set the model based on definition of the attribute.

```
@Override
    protected void populateModel(ModelNode operation, ModelNode model) throws
OperationFailedException {
         TICK.validateAndSet(operation,model);
    }
```

We then override the `performRuntime()` method to perform our runtime changes, which in this case involves installing a service into the controller at the heart of WildFly. (`AbstractAddStepHandler.performRuntime()` is similar to `AbstractBoottimeAddStepHandler.performBoottime()` in that the model is updated before runtime changes are made.

```
@Override
    protected void performRuntime(OperationContext context, ModelNode operation, ModelNode
model,
            ServiceVerificationHandler verificationHandler, List<ServiceController<?>>
newControllers)
            throws OperationFailedException {
        String suffix =
PathAddress.pathAddress(operation.get(ModelDescriptionConstants.ADDRESS)).getLastElement().getValu
long tick = TICK.resolveModelAttribute(context,model).asLong();
        TrackerService service = new TrackerService(suffix, tick);
        ServiceName name = TrackerService.createServiceName(suffix);
        ServiceController<TrackerService> controller = context.getServiceTarget()
                .addService(name, service)
                .addListener(verificationHandler)
                .setInitialMode(Mode.ACTIVE)
                .install();
        newControllers.add(controller);
    }
}
```

Since the add methods will be of the format `/subsystem=tracker/suffix=war:add(tick=1234)`, we look for the last element of the operation address, which is `war` in the example just given and use that as our suffix. We then create an instance of TrackerService and install that into the `service target` of the context and add the created `service controller` to the `newControllers` list.

The tracker service is quite simple. All services installed into WildFly must implement the `org.jboss.msc.service.Service` interface.

```
public class TrackerService implements Service<TrackerService>{
```

We then have some fields to keep the tick count and a thread which when run outputs all the deployments registered with our service.

```
private AtomicLong tick = new AtomicLong(10000);

    private Set<String> deployments = Collections.synchronizedSet(new HashSet<String>());
    private Set<String> coolDeployments = Collections.synchronizedSet(new HashSet<String>());
    private final String suffix;

    private Thread OUTPUT = new Thread() {
        @Override
        public void run() {
            while (true) {
                try {
                    Thread.sleep(tick.get());
                    System.out.println("Current deployments deployed while " + suffix + "
tracking active:\n" + deployments
                       + "\nCool: " + coolDeployments.size());
                } catch (InterruptedException e) {
                    interrupted();
                    break;
                }
            }
        }
    };

    public TrackerService(String suffix, long tick) {
        this.suffix = suffix;
        this.tick.set(tick);
    }
```

Next we have three methods which come from the `Service` interface. `getValue()` returns this service, `start()` is called when the service is started by the controller, `stop` is called when the service is stopped by the controller, and they start and stop the thread outputting the deployments.

```
@Override
    public TrackerService getValue() throws IllegalStateException, IllegalArgumentException {
        return this;
    }

    @Override
    public void start(StartContext context) throws StartException {
        OUTPUT.start();
    }

    @Override
    public void stop(StopContext context) {
        OUTPUT.interrupt();
    }
```

Next we have a utility method to create the `ServiceName` which is used to register the service in the controller.

```
public static ServiceName createServiceName(String suffix) {
        return ServiceName.JBOSS.append("tracker", suffix);
}
```

Finally we have some methods to add and remove deployments, and to set and read the `tick`. The 'cool' deployments will be explained later.

```
public void addDeployment(String name) {
        deployments.add(name);
    }

    public void addCoolDeployment(String name) {
        coolDeployments.add(name);
    }

    public void removeDeployment(String name) {
        deployments.remove(name);
        coolDeployments.remove(name);
    }

    void setTick(long tick) {
        this.tick.set(tick);
    }

    public long getTick() {
        return this.tick.get();
    }
}//TrackerService - end
```

Since we are able to add `type` children, we need a way to be able to remove them, so we create a `com.acme.corp.tracker.extension.TypeRemoveHandler`. In this case we extend `AbstractRemoveStepHandler` which takes care of removing the resource from the model so we don't need to override its `performRemove()` operationa. But we need to implement the `DescriptionProvider` method to provide the model description, and since the add handler installs the TrackerService, we need to remove that in the `performRuntime()` method.

```
public class TypeRemoveHandler extends AbstractRemoveStepHandler {

    public static final TypeRemoveHandler INSTANCE = new TypeRemoveHandler();

    private TypeRemoveHandler() {
    }


    @Override
    protected void performRuntime(OperationContext context, ModelNode operation, ModelNode
model) throws OperationFailedException {
        String suffix =
PathAddress.pathAddress(operation.get(ModelDescriptionConstants.ADDRESS)).getLastElement().getValu
ServiceName name = TrackerService.createServiceName(suffix);
        context.removeService(name);
    }

}
```

We then need a description provider for the `type` part of the model itself, so we modify TypeDefinitnion to registerAttribute

```
class TypeDefinition{
...
@Override
public void registerAttributes(ManagementResourceRegistration resourceRegistration){
    resourceRegistration.registerReadWriteAttribute(TICK, null, TrackerTickHandler.INSTANCE);
}

}
```

Then finally we need to specify that our new `type` child and associated handlers go under `/subsystem=tracker/type=*` in the model by adding registering it with the model in `SubsystemExtension.initialize()`. So we add the following just before the end of the method.

```
@Override
public void initialize(ExtensionContext context)
{
 final SubsystemRegistration subsystem = context.registerSubsystem(SUBSYSTEM_NAME, 1, 0);
 final ManagementResourceRegistration registration =
subsystem.registerSubsystemModel(TrackerSubsystemDefinition.INSTANCE);
 //Add the type child
 ManagementResourceRegistration typeChild =
registration.registerSubModel(TypeDefinition.INSTANCE);
 subsystem.registerXMLElementWriter(parser);
}
```

The above first creates a child of our main subsystem registration for the relative address `type=*`, and gets the `typeChild` registration.
To this we add the `TypeAddHandler` and `TypeRemoveHandler`.
The add variety is added under the name `add` and the remove handler under the name `remove`, and for each registered operation handler we use the handler singleton instance as both the handler parameter and as the `DescriptionProvider`.

Finally, we register `tick` as a read/write attribute, the null parameter means we don't do anything special with regards to reading it, for the write handler we supply it with an operation handler called `TrackerTickHandler`.
Registering it as a read/write attribute means we can use the `:write-attribute` operation to modify the value of the parameter, and it will be handled by `TrackerTickHandler`.

Not registering a write attribute handler makes the attribute read only.

`TrackerTickHandler` extends `AbstractWriteAttributeHandler`
directly, and so must implement its `applyUpdateToRuntime` and `revertUpdateToRuntime` method.
This takes care of model manipulation (validation, setting) but leaves us to do just to deal with what we need to do.

```
class TrackerTickHandler extends AbstractWriteAttributeHandler<Void> {

    public static final TrackerTickHandler INSTANCE = new TrackerTickHandler();

    private TrackerTickHandler() {
        super(TypeDefinition.TICK);
    }

    protected boolean applyUpdateToRuntime(OperationContext context, ModelNode operation, String
attributeName,
              ModelNode resolvedValue, ModelNode currentValue, HandbackHolder<Void>
handbackHolder) throws OperationFailedException {

        modifyTick(context, operation, resolvedValue.asLong());

        return false;
    }

    protected void revertUpdateToRuntime(OperationContext context, ModelNode operation, String
attributeName, ModelNode valueToRestore, ModelNode valueToRevert, Void handback){
        modifyTick(context, operation, valueToRestore.asLong());
    }

    private void modifyTick(OperationContext context, ModelNode operation, long value) throws
OperationFailedException {

        final String suffix =
PathAddress.pathAddress(operation.get(ModelDescriptionConstants.ADDRESS)).getLastElement().getValu
TrackerService service = (TrackerService)
context.getServiceRegistry(true).getRequiredService(TrackerService.createServiceName(suffix)).getV
service.setTick(value);
    }

}
```

The operation used to execute this will be of the form `/subsystem=tracker/type=war:write-attribute(name=tick,value=12345)` so we first get the `suffix` from the operation address, and the `tick` value from the operation parameter's `resolvedValue` parameter, and use that to update the model.

We then add a new step associated with the `RUNTIME` stage to update the tick of the TrackerService for our suffix. This is essential since the call to `context.getServiceRegistry()` will fail unless the step accessing it belongs to the `RUNTIME` stage.

When implementing `execute()`, you **must** call `context.completeStep()` when you are done.

11.11.12 Expressions

Expressions are mechanism that enables you to support variables in your attributes, for instance when you want the value of attribute to be resolved using system / environment properties.

An example expression is

```
${jboss.bind.address.management:127.0.0.1}
```

which means that the value should be taken from a system property named `jboss.bind.address.management` and if it is not defined use `127.0.0.1`.

What expression types are supported

- System properties, which are resolved using `java.lang.System.getProperty(String key)`
- Environment properties, which are resolved using `java.lang.System.getEnv(String name)`.
- Security vault expressions, resolved against the security vault configured for the server or Host Controller that needs to resolve the expression.

In all cases, the syntax for the expression is

```
${expression_to_resolve}
```

For an expression meant to be resolved against environment properties, the `expression_to_resolve` must be prefixed with `env.`. The portion after `env.` will be the name passed to `java.lang.System.getEnv(String name)`.

Security vault expressions do not support default values (i.e. the `127.0.0.1` in the `jboss.bind.address.management:127.0.0.1` example above.)

How to support expressions in subsystems

The easiest way is by using AttributeDefinition, which provides support for expressions just by using it correctly.

When we create an AttributeDefinition all we need to do is mark that is allows expressions. Here is an example how to define an attribute that allows expressions to be used.

```
SimpleAttributeDefinition MY_ATTRIBUTE =
            new SimpleAttributeDefinitionBuilder("my-attribute", ModelType.INT, true)
                    .setAllowExpression(true)
                    .setFlags(AttributeAccess.Flag.RESTART_ALL_SERVICES)
                    .setDefaultValue(new ModelNode(1))
                    .build();
```

Then later when you are parsing the xml configuration you should use the MY_ATTRIBUTE attribute definition to set the value to the management operation ModelNode you are creating.

```
....
      String attr = reader.getAttributeLocalName(i);
      String value = reader.getAttributeValue(i);
      if (attr.equals("my-attribute")) {
          MY_ATTRIBUTE.parseAndSetParameter(value, operation, reader);
      } else if (attr.equals("suffix")) {
.....
```

Note that this just helps you to properly set the value to the model node you are working on, so no need to additionally set anything to the model for this attribute. Method parseAndSetParameter parses the value that was read from xml for possible expressions in it and if it finds any it creates special model node that defines that node is of type `ModelType.EXPRESSION`.

Later in your operation handlers where you implement populateModel and have to store the value from the operation to the configuration model you also use this MY_ATTRIBUTE attribute definition.

```
@Override
 protected void populateModel(ModelNode operation, ModelNode model) throws
OperationFailedException {
        MY_ATTRIBUTE.validateAndSet(operation,model);
 }
```

This will make sure that the attribute that is stored from the operation to the model is valid and nothing is lost. It also checks the value stored in the operation `ModelNode`, and if it isn't already `ModelType.EXPRESSION`, it checks if the value is a string that contains the expression syntax. If so, the value stored in the model will be of type `ModelType.EXPRESSION`. Doing this ensures that expressions are properly handled when they appear in operations that weren't created by the subsystem parser, but are instead passed in from CLI or admin console users.

As last step we need to use the value of the attribute. This is usually needed inside of the `performRuntime` method

```
protected void performRuntime(OperationContext context, ModelNode operation, ModelNode model,
ServiceVerificationHandler verificationHandler, List<ServiceController<?>> newControllers)
throws OperationFailedException {
       ....
        final int attributeValue = MY_ATTRIBUTE.resolveModelAttribute(context,
model).asInt();
        ...

    }
```

As you can see resolving of attribute's value is not done until it is needed for use in the subsystem's runtime services. The resolved value is not stored in the configuration model, the unresolved expression is. That way we do not lose any information in the model and can assure that also marshalling is done properly, where we must marshall back the unresolved value.

Attribute definitinon also helps you with that:

```
public void writeContent(XMLExtendedStreamWriter writer, SubsystemMarshallingContext context)
throws XMLStreamException {
    ....
      MY_ATTRIBUTE.marshallAsAttribute(sessionData, writer);
      MY_OTHER_ATTRIBUTE.marshallAsElement(sessionData, false, writer);
    ...
}
```

11.11.13 Integrate with WildFly

Now that we have all the code needed for our subsystem, we can build our project by running `mvn install`

```
[kabir ~/sourcecontrol/temp/archetype-test/acme-subsystem]
$mvn install
[INFO] Scanning for projects...
[...]
main:
   [delete] Deleting:
/Users/kabir/sourcecontrol/temp/archetype-test/acme-subsystem/null1004283288
   [delete] Deleting directory
/Users/kabir/sourcecontrol/temp/archetype-test/acme-subsystem/target/module
     [copy] Copying 1 file to
/Users/kabir/sourcecontrol/temp/archetype-test/acme-subsystem/target/module/com/acme/corp/tracker/
[copy] Copying 1 file to
/Users/kabir/sourcecontrol/temp/archetype-test/acme-subsystem/target/module/com/acme/corp/tracker/
[echo] Module com.acme.corp.tracker has been created in the target/module directory. Copy to
your JBoss AS 7 installation.
[INFO] Executed tasks
[INFO]
[INFO] --- maven-install-plugin:2.3.1:install (default-install) @ acme-subsystem ---
[INFO] Installing
/Users/kabir/sourcecontrol/temp/archetype-test/acme-subsystem/target/acme-subsystem.jar to
/Users/kabir/.m2/repository/com/acme/corp/acme-subsystem/1.0-SNAPSHOT/acme-subsystem-1.0-SNAPSHOT.
Installing /Users/kabir/sourcecontrol/temp/archetype-test/acme-subsystem/pom.xml to
/Users/kabir/.m2/repository/com/acme/corp/acme-subsystem/1.0-SNAPSHOT/acme-subsystem-1.0-SNAPSHOT.
------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 5.851s
[INFO] Finished at: Mon Jul 11 23:24:58 BST 2011
[INFO] Final Memory: 7M/81M
[INFO] ------------------------------------------------------------------------
```

This will have built our project and assembled a module for us that can be used for installing it into WildFly. If you go to the `target/module` folder where you built the project you will see the module

```
$ls target/module/com/acme/corp/tracker/main/
acme-subsystem.jar  module.xml
```

The `module.xml` comes from `src/main/resources/module/main/module.xml` and is used to define your module. It says that it contains the `acme-subsystem.jar`:

```
<module xmlns="urn:jboss:module:1.0" name="com.acme.corp.tracker">
    <resources>
        <resource-root path="acme-subsystem.jar"/>
    </resources>
```

And has a default set of dependencies needed by every subsystem created. If your subsystem requires additional module dependencies you can add them here before building and installing.

```
<dependencies>
        <module name="javax.api"/>
        <module name="org.jboss.staxmapper"/>
        <module name="org.jboss.as.controller"/>
        <module name="org.jboss.as.server"/>
        <module name="org.jboss.modules"/>
        <module name="org.jboss.msc"/>
        <module name="org.jboss.logging"/>
        <module name="org.jboss.vfs"/>
    </dependencies>
</module>
```

Note that the name of the module corresponds to the directory structure containing it. Now copy the `target/module/com/acme/corp/tracker/main/` directory and its contents to `$WFLY/modules/com/acme/corp/tracker/main/` (where `$WFLY` is the root of your WildFly install).

Next we need to modify `$WFLY/standalone/configuration/standalone.xml`. First we need to add our new module to the `<extensions>` section:

```
<extensions>
        ...
        <extension module="org.jboss.as.weld"/>
        <extension module="com.acme.corp.tracker"/>
    </extensions>
```

And then we have to add our subsystem to the `<profile>` section:

```
<profile>
    ...

        <subsystem xmlns="urn:com.acme.corp.tracker:1.0">
            <deployment-types>
                <deployment-type suffix="sar" tick="10000"/>
                <deployment-type suffix="war" tick="10000"/>
            </deployment-types>
        </subsystem>
    ...
    </profile>
```

Adding this to a managed domain works exactly the same apart from in this case you need to modify `$WFLY/domain/configuration/domain.xml`.

Now start up WildFly by running `$WFLY/bin/standalone.sh` and you should see messages like these after the server has started, which means our subsystem has been added and our `TrackerService` is working:

```
15:27:33,838 INFO  [org.jboss.as] (Controller Boot Thread) JBoss AS 7.0.0.Final "Lightning"
started in 2861ms - Started 94 of 149 services (55 services are passive or on-demand)
15:27:42,966 INFO  [stdout] (Thread-8) Current deployments deployed while sar tracking active:
15:27:42,966 INFO  [stdout] (Thread-8) []
15:27:42,967 INFO  [stdout] (Thread-8) Cool: 0
15:27:42,967 INFO  [stdout] (Thread-9) Current deployments deployed while war tracking active:
15:27:42,967 INFO  [stdout] (Thread-9) []
15:27:42,967 INFO  [stdout] (Thread-9) Cool: 0
15:27:52,967 INFO  [stdout] (Thread-8) Current deployments deployed while sar tracking active:
15:27:52,967 INFO  [stdout] (Thread-8) []
15:27:52,967 INFO  [stdout] (Thread-8) Cool: 0
```

If you run the command line interface you can execute some commands to see more about the subsystem. For example

```
[standalone@localhost:9999 /] /subsystem=tracker/:read-resource-description(recursive=true,
operations=true)
```

will return a lot of information, including what we provided in the `DescriptionProviders` we created to document our subsystem.

To see the current subsystem state you can execute

```
[standalone@localhost:9999 /] /subsystem=tracker/:read-resource(recursive=true)
{
    "outcome" => "success",
    "result" => {"type" => {
        "war" => {"tick" => 10000L},
        "sar" => {"tick" => 10000L}
    }}
}
```

We can remove both the deployment types which removes them from the model:

```
[standalone@localhost:9999 /] /subsystem=tracker/type=sar:remove
{"outcome" => "success"}
[standalone@localhost:9999 /] /subsystem=tracker/type=war:remove
{"outcome" => "success"}
[standalone@localhost:9999 /] /subsystem=tracker/:read-resource(recursive=true)
{
    "outcome" => "success",
    "result" => {"type" => undefined}
}
```

You should now see the output from the `TrackerService` instances having stopped.

Now, let's add the war tracker again:

```
[standalone@localhost:9999 /] /subsystem=tracker/type=war:add
{"outcome" => "success"}
[standalone@localhost:9999 /] /subsystem=tracker/:read-resource(recursive=true)
{
    "outcome" => "success",
    "result" => {"type" => {"war" => {"tick" => 10000L}}}
}
```

and the WildFly console should show the messages coming from the war `TrackerService` again.

Now let us deploy something. You can find two maven projects for test wars already built at test1.zip and test2.zip. If you download them and extract them to `/Downloads/test1` and `/Downloads/test2`, you can see that `/Downloads/test1/target/test1.war` contains a `META-INF/cool.txt` while `/Downloads/test2/target/test2.war` does not contain that file. From CLI deploy `test1.war` first:

```
[standalone@localhost:9999 /] deploy ~/Downloads/test1/target/test1.war
'test1.war' deployed successfully.
```

And you should now see the output from the war `TrackerService` list the deployments:

```
15:35:03,712 INFO  [org.jboss.as.server.deployment] (MSC service thread 1-2) Starting deployment
of "test1.war"
15:35:03,988 INFO  [org.jboss.web] (MSC service thread 1-1) registering web context: /test1
15:35:03,996 INFO  [org.jboss.as.server.controller] (pool-2-thread-9) Deployed "test1.war"
15:35:13,056 INFO  [stdout] (Thread-9) Current deployments deployed while war tracking active:
15:35:13,056 INFO  [stdout] (Thread-9) [test1.war]
15:35:13,057 INFO  [stdout] (Thread-9) Cool: 1
```

So our `test1.war` got picked up as a 'cool' deployment. Now if we deploy `test2.war`

```
[standalone@localhost:9999 /] deploy ~/sourcecontrol/temp/archetype-test/test2/target/test2.war
'test2.war' deployed successfully.
```

You will see that deployment get picked up as well but since there is no `META-INF/cool.txt` it is not marked as a 'cool' deployment:

```
15:37:05,634 INFO  [org.jboss.as.server.deployment] (MSC service thread 1-4) Starting deployment
of "test2.war"
15:37:05,699 INFO  [org.jboss.web] (MSC service thread 1-1) registering web context: /test2
15:37:05,982 INFO  [org.jboss.as.server.controller] (pool-2-thread-15) Deployed "test2.war"
15:37:13,075 INFO  [stdout] (Thread-9) Current deployments deployed while war tracking active:
15:37:13,075 INFO  [stdout] (Thread-9) [test1.war, test2.war]
15:37:13,076 INFO  [stdout] (Thread-9) Cool: 1
```

An undeploy

```
[standalone@localhost:9999 /] undeploy test1.war
Successfully undeployed test1.war.
```

is also reflected in the `TrackerService` output:

```
15:38:47,901 INFO  [org.jboss.as.server.controller] (pool-2-thread-21) Undeployed "test1.war"
15:38:47,934 INFO  [org.jboss.as.server.deployment] (MSC service thread 1-3) Stopped deployment
test1.war in 40ms
15:38:53,091 INFO  [stdout] (Thread-9) Current deployments deployed while war tracking active:
15:38:53,092 INFO  [stdout] (Thread-9) [test2.war]
15:38:53,092 INFO  [stdout] (Thread-9) Cool: 0
```

Finally, we registered a write attribute handler for the `tick` property of the `type` so we can change the frequency

```
[standalone@localhost:9999 /] /subsystem=tracker/type=war:write-attribute(name=tick,value=1000)
{"outcome" => "success"}
```

You should now see the output from the `TrackerService` happen every second

```
15:39:43,100 INFO  [stdout] (Thread-9) Current deployments deployed while war tracking active:
15:39:43,100 INFO  [stdout] (Thread-9) [test2.war]
15:39:43,101 INFO  [stdout] (Thread-9) Cool: 0
15:39:44,101 INFO  [stdout] (Thread-9) Current deployments deployed while war tracking active:
15:39:44,102 INFO  [stdout] (Thread-9) [test2.war]
15:39:44,105 INFO  [stdout] (Thread-9) Cool: 0
15:39:45,106 INFO  [stdout] (Thread-9) Current deployments deployed while war tracking active:
15:39:45,106 INFO  [stdout] (Thread-9) [test2.war]
```

If you open `$WFLY/standalone/configuration/standalone.xml` you can see that our subsystem entry reflects the current state of the subsystem:

```
<subsystem xmlns="urn:com.acme.corp.tracker:1.0">
            <deployment-types>
                <deployment-type suffix="war" tick="1000"/>
            </deployment-types>
        </subsystem>
```

11.11.14 Parsing and marshalling of the subsystem xml

WildFly uses the Stax API to parse the xml files. This is initialized in `SubsystemExtension` by mapping our parser onto our namespace:

```
public class SubsystemExtension implements Extension {

    /** The name space used for the {@code subsystem} element */
    public static final String NAMESPACE = "urn:com.acme.corp.tracker:1.0";
    ...
    protected static final PathElement SUBSYSTEM_PATH = PathElement.pathElement(SUBSYSTEM,
SUBSYSTEM_NAME);
    protected static final PathElement TYPE_PATH = PathElement.pathElement(TYPE);

   /** The parser used for parsing our subsystem */
    private final SubsystemParser parser = new SubsystemParser();

   @Override
    public void initializeParsers(ExtensionParsingContext context) {
        context.setSubsystemXmlMapping(NAMESPACE, parser);
    }
    ...
```

We then need to write the parser. The contract is that we read our subsystem's xml and create the operations that will populate the model with the state contained in the xml. These operations will then be executed on our behalf as part of the parsing process. The entry point is the `readElement()` method.

```
public class SubsystemExtension implements Extension {

    /**
```

```
     * The subsystem parser, which uses stax to read and write to and from xml
     */
    private static class SubsystemParser implements XMLStreamConstants,
XMLElementReader<List<ModelNode>>, XMLElementWriter<SubsystemMarshallingContext> {

        /** {@inheritDoc} */
        @Override
        public void readElement(XMLExtendedStreamReader reader, List<ModelNode> list) throws
XMLStreamException {
            // Require no attributes
            ParseUtils.requireNoAttributes(reader);

            //Add the main subsystem 'add' operation
            final ModelNode subsystem = new ModelNode();
            subsystem.get(OP).set(ADD);
            subsystem.get(OP_ADDR).set(PathAddress.pathAddress(SUBSYSTEM_PATH).toModelNode());
            list.add(subsystem);

            //Read the children
            while (reader.hasNext() && reader.nextTag() != END_ELEMENT) {
                if (!reader.getLocalName().equals("deployment-types")) {
                    throw ParseUtils.unexpectedElement(reader);
                }
                while (reader.hasNext() && reader.nextTag() != END_ELEMENT) {
                    if (reader.isStartElement()) {
                        readDeploymentType(reader, list);
                    }
                }
            }
        }

        private void readDeploymentType(XMLExtendedStreamReader reader, List<ModelNode> list)
throws XMLStreamException {
            if (!reader.getLocalName().equals("deployment-type")) {
                throw ParseUtils.unexpectedElement(reader);
            }
            ModelNode addTypeOperation = new ModelNode();
            addTypeOperation.get(OP).set(ModelDescriptionConstants.ADD);

            String suffix = null;
            for (int i = 0; i < reader.getAttributeCount(); i++) {
                String attr = reader.getAttributeLocalName(i);
                String value = reader.getAttributeValue(i);
                if (attr.equals("tick")) {
                    TypeDefinition.TICK.parseAndSetParameter(value, addTypeOperation, reader);
                } else if (attr.equals("suffix")) {
                    suffix = value;
                } else {
                    throw ParseUtils.unexpectedAttribute(reader, i);
                }
            }
            ParseUtils.requireNoContent(reader);
            if (suffix == null) {
                throw ParseUtils.missingRequiredElement(reader,
Collections.singleton("suffix"));
            }

            //Add the 'add' operation for each 'type' child
```

```
            PathAddress addr = PathAddress.pathAddress(SUBSYSTEM_PATH,
PathElement.pathElement(TYPE, suffix));
            addTypeOperation.get(OP_ADDR).set(addr.toModelNode());
            list.add(addTypeOperation);
        }
        ...
```

So in the above we always create the add operation for our subsystem. Due to its address `/subsystem=tracker` defined by SUBSYSTEM_PATH this will trigger the `SubsystemAddHandler` we created earlier when we invoke `/subsystem=tracker:add`. We then parse the child elements and create an add operation for the child address for each `type` child. Since the address will for example be `/subsystem=tracker/type=sar` (defined by TYPE_PATH) and `TypeAddHandler` is registered for all `type` subaddresses the `TypeAddHandler` will get invoked for those operations. Note that when we are parsing attribute `tick` we are using definition of attribute that we defined in TypeDefintion to parse attribute value and apply all rules that we specified for this attribute, this also enables us to property support expressions on attributes.

The parser is also used to marshal the model to xml whenever something modifies the model, for which the entry point is the `writeContent()` method:

```
private static class SubsystemParser implements XMLStreamConstants,
XMLElementReader<List<ModelNode>>, XMLElementWriter<SubsystemMarshallingContext> {
        ...
        /** {@inheritDoc} */
        @Override
        public void writeContent(final XMLExtendedStreamWriter writer, final
SubsystemMarshallingContext context) throws XMLStreamException {
            //Write out the main subsystem element
            context.startSubsystemElement(TrackerExtension.NAMESPACE, false);
            writer.writeStartElement("deployment-types");
            ModelNode node = context.getModelNode();
            ModelNode type = node.get(TYPE);
            for (Property property : type.asPropertyList()) {

                //write each child element to xml
                writer.writeStartElement("deployment-type");
                writer.writeAttribute("suffix", property.getName());
                ModelNode entry = property.getValue();
                TypeDefinition.TICK.marshallAsAttribute(entry, true, writer);
                writer.writeEndElement();
            }
            //End deployment-types
            writer.writeEndElement();
            //End subsystem
            writer.writeEndElement();
        }
    }
```

Then we have to implement the `SubsystemDescribeHandler` which translates the current state of the model into operations similar to the ones created by the parser. The `SubsystemDescribeHandler` is only used when running in a managed domain, and is used when the host controller queries the domain controller for the configuration of the profile used to start up each server. In our case the `SubsystemDescribeHandler` adds the operation to add the subsystem and then adds the operation to add each `type` child. Since we are using ResourceDefinitinon for defining subsystem all that is generated for us, but if you want to customize that you can do it by implementing it like this.

```
private static class SubsystemDescribeHandler implements OperationStepHandler,
DescriptionProvider {
        static final SubsystemDescribeHandler INSTANCE = new SubsystemDescribeHandler();

        public void execute(OperationContext context, ModelNode operation) throws
OperationFailedException {
            //Add the main operation
            context.getResult().add(createAddSubsystemOperation());

            //Add the operations to create each child

            ModelNode node = context.readModel(PathAddress.EMPTY_ADDRESS);
            for (Property property : node.get("type").asPropertyList()) {

                ModelNode addType = new ModelNode();
                addType.get(OP).set(ModelDescriptionConstants.ADD);
                PathAddress addr = PathAddress.pathAddress(SUBSYSTEM_PATH,
PathElement.pathElement("type", property.getName()));
                addType.get(OP_ADDR).set(addr.toModelNode());
                if (property.getValue().hasDefined("tick")) {
                   TypeDefinition.TICK.validateAndSet(property,addType);
                }
                context.getResult().add(addType);
            }
            context.completeStep();
        }


}
```

Testing the parsers

Changes to tests between 7.0.0 and 7.0.1

The testing framework was moved from the archetype into the core JBoss AS 7 sources between JBoss AS 7.0.0 and JBoss AS 7.0.1, and has been improved upon and is used internally for testing JBoss AS 7's subsystems. The differences between the two versions is that in 7.0.0.Final the testing framework is bundled with the code generated by the archetype (in a sub-package of the package specified for your subsystem, e.g. `com.acme.corp.tracker.support`), and the test extends the `AbstractParsingTest` class.

From 7.0.1 the testing framework is now brought in via the `org.jboss.as:jboss-as-subsystem-test` maven artifact, and the test's superclass is `org.jboss.as.subsystem.test.AbstractSubsystemTest`. The concepts are the same but more and more functionality will be available as JBoss AS 7 is developed.

Now that we have modified our parsers we need to update our tests to reflect the new model. There are currently three tests testing the basic functionality, something which is a lot easier to debug from your IDE before you plug it into the application server. We will talk about these tests in turn and they all live in `com.acme.corp.tracker.extension.SubsystemParsingTestCase`. `SubsystemParsingTestCase` extends `AbstractSubsystemTest` which does a lot of the setup for you and contains utility methods for verifying things from your test. See the javadoc of that class for more information about the functionality available to you. And by all means feel free to add more tests for your subsystem, here we are only testing for the best case scenario while you will probably want to throw in a few tests for edge cases.

The first test we need to modify is `testParseSubsystem()`. It tests that the parsed xml becomes the expected operations that will be parsed into the server, so let us tweak this test to match our subsystem. First we tell the test to parse the xml into operations

```
@Test
    public void testParseSubsystem() throws Exception {
        //Parse the subsystem xml into operations
        String subsystemXml =
                "<subsystem xmlns=\"" + SubsystemExtension.NAMESPACE + "\">" +
                "   <deployment-types>" +
                "       <deployment-type suffix=\"tst\" tick=\"12345\"/>" +
                "   </deployment-types>" +
                "</subsystem>";
        List<ModelNode> operations = super.parse(subsystemXml);
```

There should be one operation for adding the subsystem itself and an operation for adding the `deployment-type`, so check we got two operations

```
///Check that we have the expected number of operations
        Assert.assertEquals(2, operations.size());
```

Now check that the first operation is `add` for the address `/subsystem=tracker`:

```
//Check that each operation has the correct content
        //The add subsystem operation will happen first
        ModelNode addSubsystem = operations.get(0);
        Assert.assertEquals(ADD, addSubsystem.get(OP).asString());
        PathAddress addr = PathAddress.pathAddress(addSubsystem.get(OP_ADDR));
        Assert.assertEquals(1, addr.size());
        PathElement element = addr.getElement(0);
        Assert.assertEquals(SUBSYSTEM, element.getKey());
        Assert.assertEquals(SubsystemExtension.SUBSYSTEM_NAME, element.getValue());
```

Then check that the second operation is `add` for the address `/subsystem=tracker`, and that `12345` was picked up for the value of the `tick` parameter:

```
//Then we will get the add type operation
        ModelNode addType = operations.get(1);
        Assert.assertEquals(ADD, addType.get(OP).asString());
        Assert.assertEquals(12345, addType.get("tick").asLong());
        addr = PathAddress.pathAddress(addType.get(OP_ADDR));
        Assert.assertEquals(2, addr.size());
        element = addr.getElement(0);
        Assert.assertEquals(SUBSYSTEM, element.getKey());
        Assert.assertEquals(SubsystemExtension.SUBSYSTEM_NAME, element.getValue());
        element = addr.getElement(1);
        Assert.assertEquals("type", element.getKey());
        Assert.assertEquals("tst", element.getValue());
    }
```

The second test we need to modify is `testInstallIntoController()` which tests that the xml installs properly into the controller. In other words we are making sure that the `add` operations we created earlier work properly. First we create the xml and install it into the controller. Behind the scenes this will parse the xml into operations as we saw in the last test, but it will also create a new controller and boot that up using the created operations

```
@Test
    public void testInstallIntoController() throws Exception {
        //Parse the subsystem xml and install into the controller
        String subsystemXml =
                "<subsystem xmlns=\"" + SubsystemExtension.NAMESPACE + "\">" +
                "   <deployment-types>" +
                "       <deployment-type suffix=\"tst\" tick=\"12345\"/>" +
                "   </deployment-types>" +
                "</subsystem>";
        KernelServices services = super.installInController(subsystemXml);
```

The returned `KernelServices` allow us to execute operations on the controller, and to read the whole model.

```
//Read the whole model and make sure it looks as expected
        ModelNode model = services.readWholeModel();
        //Useful for debugging :-)
        //System.out.println(model);
```

Now we make sure that the structure of the model within the controller has the expected format and values

```
Assert.assertTrue(model.get(SUBSYSTEM).hasDefined(SubsystemExtension.SUBSYSTEM_NAME));
        Assert.assertTrue(model.get(SUBSYSTEM,
SubsystemExtension.SUBSYSTEM_NAME).hasDefined("type"));
        Assert.assertTrue(model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME,
"type").hasDefined("tst"));
        Assert.assertTrue(model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME, "type",
"tst").hasDefined("tick"));
        Assert.assertEquals(12345, model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME,
"type", "tst", "tick").asLong());
    }
```

The last test provided is called `testParseAndMarshalModel()`. It's main purpose is to make sure that our `SubsystemParser.writeContent()` works as expected. This is achieved by starting a controller in the same way as before

```
@Test
    public void testParseAndMarshalModel() throws Exception {
        //Parse the subsystem xml and install into the first controller
        String subsystemXml =
                "<subsystem xmlns=\"" + SubsystemExtension.NAMESPACE + "\">" +
                "   <deployment-types>" +
                "       <deployment-type suffix=\"tst\" tick=\"12345\"/>" +
                "   </deployment-types>" +
                "</subsystem>";
        KernelServices servicesA = super.installInController(subsystemXml);
```

Now we read the model and the xml that was persisted from the first controller, and use that xml to start a second controller

```
//Get the model and the persisted xml from the first controller
        ModelNode modelA = servicesA.readWholeModel();
        String marshalled = servicesA.getPersistedSubsystemXml();

        //Install the persisted xml from the first controller into a second controller
        KernelServices servicesB = super.installInController(marshalled);
```

Finally we read the model from the second controller, and make sure that the models are identical by calling `compare()` on the test superclass.

```
ModelNode modelB = servicesB.readWholeModel();

        //Make sure the models from the two controllers are identical
        super.compare(modelA, modelB);
    }
```

We then have a test that needs no changing from what the archetype provides us with. As we have seen before we start a controller

```
@Test
    public void testDescribeHandler() throws Exception {
        //Parse the subsystem xml and install into the first controller
        String subsystemXml =
                "<subsystem xmlns=\"" + SubsystemExtension.NAMESPACE + "\">" +
                "</subsystem>";
        KernelServices servicesA = super.installInController(subsystemXml);
```

We then call `/subsystem=tracker:describe` which outputs the subsystem as operations needed to reach the current state (Done by our `SubsystemDescribeHandler`)

```
//Get the model and the describe operations from the first controller
        ModelNode modelA = servicesA.readWholeModel();
        ModelNode describeOp = new ModelNode();
        describeOp.get(OP).set(DESCRIBE);
        describeOp.get(OP_ADDR).set(
                PathAddress.pathAddress(
                        PathElement.pathElement(SUBSYSTEM,
SubsystemExtension.SUBSYSTEM_NAME)).toModelNode());
        List<ModelNode> operations =
super.checkResultAndGetContents(servicesA.executeOperation(describeOp)).asList();
```

Then we create a new controller using those operations

```
//Install the describe options from the first controller into a second controller
        KernelServices servicesB = super.installInController(operations);
```

And then we read the model from the second controller and make sure that the two subsystems are identical ModelNode modelB = servicesB.readWholeModel();

```
//Make sure the models from the two controllers are identical
        super.compare(modelA, modelB);

    }
```

To test the removal of the the subsystem and child resources we modify the `testSubsystemRemoval()` test provided by the archetype:

```
/**
     * Tests that the subsystem can be removed
     */
    @Test
    public void testSubsystemRemoval() throws Exception {
        //Parse the subsystem xml and install into the first controller
```

We provide xml for the subsystem installing a child, which in turn installs a TrackerService

```
String subsystemXml =
                "<subsystem xmlns=\"" + SubsystemExtension.NAMESPACE + "\">" +
                "   <deployment-types>" +
                "       <deployment-type suffix=\"tst\" tick=\"12345\"/>" +
                "   </deployment-types>" +
                "</subsystem>";
        KernelServices services = super.installInController(subsystemXml);
```

Having installed the xml into the controller we make sure the TrackerService is there

```
//Sanity check to test the service for 'tst' was there
        services.getContainer().getRequiredService(TrackerService.createServiceName("tst"));
```

This call from the subsystem test harness will call remove for each level in our subsystem, children first and validate
that the subsystem model is empty at the end.

```
//Checks that the subsystem was removed from the model
        super.assertRemoveSubsystemResources(services);
```

Finally we check that all the services were removed by the remove handlers

```
//Check that any services that were installed were removed here
        try {
            services.getContainer().getRequiredService(TrackerService.createServiceName("tst"));
            Assert.fail("Should have removed services");
        } catch (Exception expected) {
        }
    }
```

For good measure let us throw in another test which adds a `deployment-type` and also changes its attribute at runtime. So first of all boot up the controller with the same xml we have been using so far

```
@Test
    public void testExecuteOperations() throws Exception {
        String subsystemXml =
                "<subsystem xmlns=\"" + SubsystemExtension.NAMESPACE + "\">" +
                "   <deployment-types>" +
                "       <deployment-type suffix=\"tst\" tick=\"12345\"/>" +
                "   </deployment-types>" +
                "</subsystem>";
        KernelServices services = super.installInController(subsystemXml);
```

Now create an operation which does the same as the following CLI command `/subsystem=tracker/type=foo:add(tick=1000)`

```
//Add another type
        PathAddress fooTypeAddr = PathAddress.pathAddress(
                PathElement.pathElement(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME),
                PathElement.pathElement("type", "foo"));
        ModelNode addOp = new ModelNode();
        addOp.get(OP).set(ADD);
        addOp.get(OP_ADDR).set(fooTypeAddr.toModelNode());
        addOp.get("tick").set(1000);
```

Execute the operation and make sure it was successful

```
ModelNode result = services.executeOperation(addOp);
        Assert.assertEquals(SUCCESS, result.get(OUTCOME).asString());
```

Read the whole model and make sure that the original data is still there (i.e. the same as what was done by `testInstallIntoController()`

```
ModelNode model = services.readWholeModel();
        Assert.assertTrue(model.get(SUBSYSTEM).hasDefined(SubsystemExtension.SUBSYSTEM_NAME));
        Assert.assertTrue(model.get(SUBSYSTEM,
SubsystemExtension.SUBSYSTEM_NAME).hasDefined("type"));
        Assert.assertTrue(model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME,
"type").hasDefined("tst"));
        Assert.assertTrue(model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME, "type",
"tst").hasDefined("tick"));
        Assert.assertEquals(12345, model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME,
"type", "tst", "tick").asLong());
```

Then make sure our new `type` has been added:

```
Assert.assertTrue(model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME,
"type").hasDefined("foo"));
        Assert.assertTrue(model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME, "type",
"foo").hasDefined("tick"));
        Assert.assertEquals(1000, model.get(SUBSYSTEM, SubsystemExtension.SUBSYSTEM_NAME,
"type", "foo", "tick").asLong());
```

Then we call `write-attribute` to change the `tick` value of `/subsystem=tracker/type=foo`:

```
//Call write-attribute
        ModelNode writeOp = new ModelNode();
        writeOp.get(OP).set(WRITE_ATTRIBUTE_OPERATION);
        writeOp.get(OP_ADDR).set(fooTypeAddr.toModelNode());
        writeOp.get(NAME).set("tick");
        writeOp.get(VALUE).set(3456);
        result = services.executeOperation(writeOp);
        Assert.assertEquals(SUCCESS, result.get(OUTCOME).asString());
```

To give you exposure to other ways of doing things, now instead of reading the whole model to check the attribute, we call `read-attribute` instead, and make sure it has the value we set it to.

```
//Check that write attribute took effect, this time by calling read-attribute instead of reading
the whole model
        ModelNode readOp = new ModelNode();
        readOp.get(OP).set(READ_ATTRIBUTE_OPERATION);
        readOp.get(OP_ADDR).set(fooTypeAddr.toModelNode());
        readOp.get(NAME).set("tick");
        result = services.executeOperation(readOp);
        Assert.assertEquals(3456, checkResultAndGetContents(result).asLong());
```

Since each `type` installs its own copy of `TrackerService`, we get the `TrackerService` for `type=foo` from the service container exposed by the kernel services and make sure it has the right value

```
TrackerService service =
(TrackerService)services.getContainer().getService(TrackerService.createServiceName("foo")).getVal
Assert.assertEquals(3456, service.getTick());
    }
```

TypeDefinition.TICK.

11.12 Key Interfaces and Classes Relevant to Extension Developers

In the first major section of this guide, we provided an example of how to implement an extension to the AS. The emphasis there was learning by doing. In this section, we'll focus a bit more on the major WildFly interfaces and classes that most are relevant to extension developers. The best way to learn about these interfaces and classes in detail is to look at their javadoc. What we'll try to do here is provide a brief introduction of the key items and how they relate to each other.

Before digging into this section, readers are encouraged to read the "Core Management Concepts" section of the Admin Guide.

11.12.1 Extension Interface

The `org.jboss.as.controller.Extension` interface is the hook by which your extension to the AS kernel is able to integrate with the AS. During boot of the AS, when the `<extension>` element in the AS's xml configuration file naming your extension is parsed, the JBoss Modules module named in the element's name attribute is loaded. The standard JDK `java.lang.ServiceLoader` mechanism is then used to load your module's implementation of this interface.

The function of an `Extension` implementation is to register with the core AS the management API, xml parsers and xml marshallers associated with the extension module's subsystems. An `Extension` can register multiple subsystems, although the usual practice is to register just one per extension.

Once the `Extension` is loaded, the core AS will make two invocations upon it:

- `void initializeParsers(ExtensionParsingContext context)`

When this is invoked, it is the `Extension` implementation's responsibility to initialize the XML parsers for this extension's subsystems and register them with the given `ExtensionParsingContext`. The parser's job when it is later called is to create `org.jboss.dmr.ModelNode` objects representing WildFly management API operations needed make the AS's running configuration match what is described in the xml. Those management operation `ModelNode` s are added to a list passed in to the parser.

A parser for each version of the xml schema used by a subsystem should be registered. A well behaved subsystem should be able to parse any version of its schema that it has ever published in a final release.

- `void initialize(ExtensionContext context)`

When this is invoked, it is the `Extension` implementation's responsibility to register with the core AS the management API for its subsystems, and to register the object that is capable of marshalling the subsystem's in-memory configuration back to XML. Only one XML marshaller is registered per subsystem, even though multiple XML parsers can be registered. The subsystem should always write documents that conform to the latest version of its XML schema.

The registration of a subsystem's management API is done via the `ManagementResourceRegistration` interface. Before discussing that interface in detail, let's describe how it (and the related `Resource` interface) relate to the notion of managed resources in the AS.

11.12.2 WildFly Managed Resources

Each subsystem is responsible for managing one or more management resources. The conceptual characteristics of a management resource are covered in some detail in the Admin Guide; here we'll just summarize the main points. A management resource has

- An ***address*** consisting of a list of key/value pairs that uniquely identifies a resource
- Zero or more ***attributes***, the value of which is some sort of `org.jboss.dmr.ModelNode`
- Zero or more supported ***operations***. An operation has a string name and zero or more parameters, each of which is a key/value pair where the key is a string naming the parameter and the value is some sort of `ModelNode`
- Zero or more ***children***, each of which in turn is a managed resource

The implementation of a managed resource is somewhat analogous to the implementation of a Java object. A managed resource will have a "type", which encapsulates API information about that resource and logic used to implement that API. And then there are actual instances of the resource, which primarily store data representing the current state of a particular resource. This is somewhat analogous to the "class" and "object" notions in Java.

A managed resource's type is encapsulated by the `org.jboss.as.controller.registry.ManagementResourceRegistration` the core AS creates when the type is registered. The data for a particular instance is encapsulated in an implementation of the `org.jboss.as.controller.registry.Resource` interface.

11.12.3 ManagementResourceRegistration Interface

In the Java analogy used above, the `ManagementResourceRegistration` is analogous to the "class", while the `Resource` discussed below is analogous to an instance of that class.

A `ManagementResourceRegistration` represents the specification for a particular managed resource type. All resources whose address matches the same pattern will be of the same type, specified by the type's `ManagementResourceRegistration`. The MRR encapsulates:

- A `PathAddress` showing the address pattern that matches resources of that type. This `PathAddress` can and typically does involve wildcards in the value of one or more elements of the address. In this case there can be more than one instance of the type, i.e. different `Resource` instances.
- Definition of the various attributes exposed by resources of this type, including the `OperationStepHandler` implementations used for reading and writing the attribute values.
- Definition of the various operations exposed by resources of this type, including the `OperationStepHandler` implementations used for handling user invocations of those operations.
- Definition of child resource types. `ManagementResourceRegistration` instances form a tree.
- Definition of management notifications emitted by resources of this type.
- Definition of capabilities provided by resources of this type.
- Definition of RBAC access constraints that should be applied by the management kernel when authorizing operations against resources of this type.
- Whether the resource type is an alias to another resource type, and if so information about that relationship. Aliases are primarily used to preserve backwards compatibility of the management API when the location of a given type of resources is moved in a newer release.

The `ManagementResourceRegistration` interface is a subinterface of `ImmutableManagementResourceRegistration`, which provides a read-only view of the information encapsulated by the MRR. The MRR subinterface adds the methods needed for registering the attributes, operations, children, etc.

Extension developers do not directly instantiate an MRR. Instead they create a `ResourceDefinition` for the root resource type for each subsystem, and register it with the `ExtensionContext` passed in to their `Extension` implementation's `initialize` method:

```
public void initialize(ExtensionContext context) {
        SubsystemRegistration subsystem = context.registerSubsystem(SUBSYSTEM_NAME,
CURRENT_VERSION);
        subsystem.registerXMLElementWriter(getOurXmlWriter());
        ResourceDefinition rd = getOurSubsystemDefinition();
        ManagementResourceRegistration mrr = subsystem.registerSubsystemModel(rd));
    }
```

The kernel uses the provided `ResourceDefinition` to construct a `ManagementResourceRegistration` and then passes that MRR to the various `registerXXX` methods implemented by the `ResourceDefinition`, giving it the change to record the resource type's attributes, operations and children.

11.12.4 ResourceDefinition Interface

An implementation of `ResourceDefinition` is the primary class used by an extension developer when defining a managed resource type. It provides basic information about the type, exposes a `DescriptionProvider` used to generate a DMR description of the type, and implements callbacks the kernel can invoke when building up the `ManagementResourceRegistration` to ask for registration of definitions of attributes, operations, children, notifications and capabilities.

Almost always an extension author will create their `ResourceDefinition` by creating a subclass of the `org.jboss.as.controller.SimpleResourceDefinition` class or of its `PersistentResourceDefinition` subclass. Both of these classes have constructors that take a `Parameters` object, which is a simple builder class to use to provide most of the key information about the resource type. The extension-specific subclass would then take responsibility for any additional behavior needed by overriding the `registerAttributes`, `registerOperations`, `registerNotifications` and `registerChildren` callbacks to do whatever is needed beyond what is provided by the superclasses.

For example, to add a writable attribute:

```
@Override
    public void registerAttributes(ManagementResourceRegistration resourceRegistration) {
        super.registerAttributes(resourceRegistration);
        // Now we register the 'foo' attribute
        AttributeDefinition ad = FOO; // constant declared elsewhere
        OperationStepHandler writeHandler = new FooWriteAttributeHandler();
        resourceRegistration.registerReadWriteHandler(ad, null, writeHandler); // null read
handler means use default read handling
    }
```

To register a custom operation:

```
@Override
    public void registerOperations(ManagementResourceRegistration resourceRegistration) {
        super.registerOperations(resourceRegistration);
        // Now we register the 'foo-bar' custom operation
        OperationDefinition od = FooBarOperationStepHandler.getDefinition();
        OperationStepHandler osh = new FooBarOperationStepHandler();
        resourceRegistration.registerOperationHandler(od, osh);
    }
```

To register a child resource type:

```
@Override
    public void registerChildren(ManagementResourceRegistration resourceRegistration) {
        super.registerChildren(resourceRegistration);
        // Now we register the 'baz=*' child type
        ResourceDefinition rd = new BazResourceDefinition();
        resourceRegistration.registerSubmodel(rd);
    }
```

ResourceDescriptionResolver

One of the things a `ResourceDefinition` must be able to do is provide a `DescriptionProvider` that provides a proper DMR description of the resource to use as the output for the standard `read-resource-description` management operation. Since you are almost certainly going to be using one of the standard `ResourceDefinition` implementations like `SimpleResourceDefinition`, the creation of this `DescriptionProvider` is largely handled for you. The one thing that is not handled for you is providing the localized free form text descriptions of the various attributes, operations, operation parameters, child types, etc used in creating the resource description.

For this you must provide an implementation of the `ResourceDescriptionResolver` interface, typically passed to the `Parameters` object provided to the `SimpleResourceDefinition` constructor. This interface has various methods that are invoked when a piece of localized text description is needed.

Almost certainly you'll satisfy this requirement by providing an instance of the `StandardResourceDescriptionResolver` class.

`StandardResourceDescriptionResolver` uses a `ResourceBundle` to load text from a properties file available on the classpath. The keys in the properties file must follow patterns expected by `StandardResourceDescriptionResolver`. See the `StandardResourceDescriptionResolver` javadoc for further details.

The biggest task here is to create the properties file and add the text descriptions. A text description must be provided for everything. The typical thing to do is to store this properties file in the same package as your `Extension` implementation, in a file named `LocalDescriptions.properties`.

11.12.5 AttributeDefinition Class

The `AttributeDefinition` class is used to create the static definition of one of a managed resource's attributes. It's a bit poorly named though, because the same interface is used to define the details of parameters to operations, and to define fields in the result of of operations.

The definition includes all the static information about the attribute/operation parameter/result field, e.g. the DMR `ModelType` of its value, whether its presence is required, whether it supports expressions, etc. See Description of the Management Model for a description of the metadata available. Almost all of this comes from the `AttributeDefinition`.

Besides basic metadata, the `AttributeDefinition` can also hold custom logic the kernel should use when dealing with the attribute/operation parameter/result field. For example, a `ParameterValidator` to use to perform special validation of values (beyond basic things like DMR type checks and defined/undefined checks), or an `AttributeParser` or `AttributeMarshaller` to use to perform customized parsing from and marshaling to XML.

WildFly Core's `controller` module provides a number of subclasses of `AttributeDefinition` used for the usual kinds of attributes. For each there is an associated builder class which you should use to build the `AttributeDefinition`. Most commonly used are `SimpleAttributeDefinition`, built by the associated `SimpleAttributeDefinitionBuilder`. This is used for attributes whose values are analogous to java primitives, `String` or byte[]. For collections, there are various subclasses of `ListAttributeDefinition` and `MapAttributeDefinition`. All have a `Builder` inner class. For complex attributes, i.e. those with a fixed set of fully defined fields, use `ObjectTypeAttributeDefinition`. (Each field in the complex type is itself specified by an `AttributeDefinition`.) Finally there's `ObjectListAttributeDefinition` and `ObjectMapAttributeDefinition` for lists whose elements are complex types and maps whose values are complex types respectively.

Here's an example of creating a simple attribute definition with extra validation of the range of allowed values:

```
static final AttributeDefinition QUEUE_LENGTH = new
SimpleAttributeDefinitionBuilder("queue-length", ModelType.INT)
                .setRequired(true)
                .setAllowExpression(true)
                .setValidator(new IntRangeValidator(1, Integer.MAX_VALUE))
                .setRestartAllServices() // means modification after resource add puts the
server in reload-required
                .build();
```

Via a bit of dark magic, the kernel knows that the `IntRangeValidator` defined here is a reliable source of information on min and max values for the attribute, so when creating the `read-resource-description` output for the attribute it will use it and output `min` and `max` metadata. For STRING attributes, `StringLengthValidator` can also be used, and the kernel will see this and provide `min-length` and `max-length` metadata. In both cases the kernel is checking for the presence of a `MinMaxValidator` and if found it provides the appropriate metadata based on the type of the attribute.

Use `EnumValidator` to restrict a STRING attribute's values to a set of legal values:

```
static final SimpleAttributeDefinition TIME_UNIT = new SimpleAttributeDefinitionBuilder("unit",
ModelType.STRING)
            .setRequired(true)
            .setAllowExpression(true)
            .setValidator(new EnumValidator<TimeUnit>(TimeUnit.class))
            .build();
```

`EnumValidator` is an implementation of `AllowedValuesValidator` that works with Java enums. You can use other implementations or write your own to do other types of restriction to certain values.

Via a bit of dark magic similar to what is done with `MinMaxValidator`, the kernel recognizes the presence of an `AllowedValuesValidator` and uses it to seed the `allowed-values` metadata in `read-resource-description` output.

Key Uses of AttributeDefinition

Your `AttributeDefinition` instances will be some of the most commonly used objects in your extension code. Following are the most typical uses. In each of these examples assume there is a `SimpleAttributeDefinition` stored in a constant FOO_AD that is available to the code. Typically FOO_AD would be a constant in the relevant `ResourceDefinition` implementation class. Assume FOO_AD represents an INT attribute.

Note that for all of these cases except for "Use in Extracting Data from the Configuration Model for Use in Runtime Services" there may be utility code that handles this for you. For example `PersistentResourceXMLParser` can handle the XML cases, and `AbstractAddStepHandler` can handle the "Use in Storing Data Provided by the User to the Configuration Model" case.

Use in XML Parsing

Here we have your extension's implementation of `XMLElementReader<List<ModelNode>>` that is being used to parse the xml for your subsystem and add `ModelNode` operations to the list that will be used to boot the server.

```
@Override
    public void readElement(final XMLExtendedStreamReader reader, final List<ModelNode>
operationList) throws XMLStreamException {
        // Create a node for the op to add our subsystem
        ModelNode addOp = new ModelNode();
        addOp.get("address").add("subsystem", "mysubsystem");
        addOp.get("operation").set("add");
        operationList.add(addOp);

        for (int i = 0; i < reader.getAttributeCount(); i++) {
            final String value = reader.getAttributeValue(i);
            final String attribute = reader.getAttributeLocalName(i);
            if (FOO_AD.getXmlName().equals(attribute) {
                FOO_AD.parseAndSetParameter(value, addOp, reader);
            } else ....
        }

        ... more parsing
    }
```

Note that the parsing code has deliberately been abbreviated. The key point is the `parseAndSetParameter` call. FOO_AD will validate the `value` read from XML, throwing an XMLStreamException with a useful message if invalid, including a reference to the current location of the `reader`. If valid, `value` will be converted to a DMR `ModelNode` of the appropriate type and stored as a parameter field of `addOp`. The name of the parameter will be what `FOO_AD.getName()` returns.

If you use `PersistentResourceXMLParser` this parsing logic is handled for you and you don't need to write it yourself.

Use in Storing Data Provided by the User to the Configuration Model

Here we illustrate code in an `OperationStepHandler` that extracts a value from a user-provided `operation` and stores it in the internal model:

```
@Override
    public void execute(OperationContext context, ModelNode operation) throws
OperationFailedException {
        // Get the Resource targeted by this operation
        Resource resource = context.readResourceForUpdate(PathAddress.EMPTY_ADDRESS);
        ModelNode model = resource.getModel();
        // Store the value of any 'foo' param to the model's 'foo' attribute
        FOO_AD.validateAndSet(operation, model);

        ... do other stuff
    }
```

As the name implies `validateAndSet` will validate the value in `operation` before setting it. A validation failure will result in an `OperationFailedException` with an appropriate message, which the kernel will use to provide a failure response to the user.

Note that `validateAndSet` will not perform expression resolution. Expression resolution is not appropriate at this stage, when we are just trying to store data to the persistent configuration model. However, it will check for expressions and fail validation if found and FOO_AD wasn't built with `setAllowExpressions(true)`.

This work of storing data to the configuration model is usually done in handlers for the `add` and `write-attribute` operations. If you base your handler implementations on the standard classes provided by WildFly Core, this part of the work will be handled for you.

Use in Extracting Data from the Configuration Model for Use in Runtime Services

This is the example you are most likely to use in your code, as this is where data needs to be extracted from the configuration model and passed to your runtime services. What your services need is custom, so there's no utility code we provide.

Assume as part of `... do other stuff` in the last example that your handler adds a step to do further work once operation execution proceeds to RUNTIME state (see Operation Execution and the `OperationContext` for more on what this means):

```
context.addStep(new OperationStepHandler() {
            @Override
            public void execute(OperationContext context, ModelNode operation) throws
OperationFailedException {

                // Get the Resource targetted by this operation
                Resource resource = context.readResource(PathAddress.EMPTY_ADDRESS);
                ModelNode model = resource.getModel();
                // Extract the value of the 'foo' attribute from the model
                int foo = FOO_AD.resolveModelAttribute(context, model).asInt();

                Service<XyZ> service = new MyService(foo);

                ... do other stuff, like install 'service' with MSC
    }
        }, Stage.RUNTIME);
```

Use `resolveModelAttribute` to extract data from the model. It does a number of things:

- reads the value from the model
- if it's an expression and expressions are supported, resolves it
- if it's undefined and undefined is allowed but FOO_AD was configured with a default value, uses the default value
- validates the result of that (which is how we check that expressions resolve to legal values), throwing OperationFailedException with a useful message if invalid
- returns that as a `ModelNode`

If when you built FOO_AD you configured it such that the user must provide a value, or if you configured it with a default value, then you know the return value of `resolveModelAttribute` will be a defined `ModelNode`. Hence you can safely perform type conversions with it, as we do in the example above with the call to `asInt()`. If FOO_AD was configured such that it's possible that the attribute won't have a defined value, you need to guard against that, e.g.:

```
ModelNode node = FOO_AD.resolveModelAttribute(context, model);
    Integer foo = node.isDefined() ? node.asInt() : null;
```

Use in Marshaling Configuration Model Data to XML

Your `Extension` must register an `XMLElementWriter<SubsystemMarshallingContext>` for each subsystem. This is used to marshal the subsystem's configuration to XML. If you don't use `PersistentResourceXMLParser` for this you'll need to write your own marshaling code, and `AttributeDefinition` will be used.

```
@Override
    public void writeContent(XMLExtendedStreamWriter writer, SubsystemMarshallingContext
context) throws XMLStreamException {
        context.startSubsystemElement(Namespace.CURRENT.getUriString(), false);

        ModelNode subsystemModel = context.getModelNode();
        // we persist foo as an xml attribute
        FOO_AD.marshalAsAttribute(subsystemModel, writer);
        // We also have a different attribute that we marshal as an element
        BAR_AD.marshalAsElement(subsystemModel, writer);
    }
```

The `SubsystemMarshallingContext` provides a `ModelNode` that represents the entire resource tree for the subsystem (including child resources). Your `XMLElementWriter` should walk through that model, using `marshalAsAttribute` or `marshalAsElement` to write the attributes in each resource. If the model includes child node trees that represent child resources, create child xml elements for those and continue down the tree.

11.12.6 OperationDefinition and OperationStepHandler Interfaces

`OperationDefinition` defines an operation, particularly its name, its parameters and the details of any result value, with `AttributeDefinition` instances used to define the parameters and result details. The `OperationDefinition` is used to generate the `read-operation-description` output for the operation, and in some cases is also used by the kernel to decide details as to how to execute the operation.

Typically `SimpleOperationDefinitionBuilder` is used to create an `OperationDefinition`. Usually you only need to create an `OperationDefinition` for custom operations. For the common `add` and `remove` operations, if you provide minimal information about your handlers to your `SimpleResourceDefinition` implementation via the `Parameters` object passed to its constructor, then `SimpleResourceDefinition` can generate a correct `OperationDefinition` for those operations.

The `OperationStepHandler` is what contains the actual logic for doing what the user requests when they invoke an operation. As its name implies, each OSH is responsible for doing one step in the overall sequence of things necessary to give effect to what the user requested. One of the things an OSH can do is add other steps, with the result that an overall operation can involve a great number of OSHs executing. (See Operation Execution and the `OperationContext` for more on this.)

Each OSH is provided in its `execute` method with a reference to the `OperationContext` that is controlling the overall operation, plus an `operation` `ModelNode` that represents the operation that particular OSH is being asked to deal with. The `operation` node will be of `ModelType.OBJECT` with the following key/value pairs:

- a key named `operation` with a value of `ModelType.STRING` that represents the name of the operation. Typically an OSH doesn't care about this information as it is written for an operation with a particular name and will only be invoked for that operation.
- a key named `address` with a value of `ModelType.LIST` with list elements of `ModelType.PROPERTY`. This value represents the address of the resource the operation targets. If this key is not present or the value is undefined or an empty list, the target is the root resource. Typically an OSH doesn't care about this information as it can more efficiently get the address from the `OperationContext` via its `getCurrentAddress()` method.
- other key/value pairs that represent parameters to the operation, with the key the name of the parameter. This is the main information an OSH would want from the `operation` node.

There are a variety of situations where extension code will instantiate an `OperationStepHandler`

- When registering a writable attribute with a `ManagementResourceRegistration` (typically in an implementation of `ResourceDefinition.registerAttributes`), an OSH must be provided to handle the `write-attribute` operation.
- When registering a read-only or read-write attribute that needs special handling of the `read-attribute` operation, an OSH must be provided.
- When registering a metric attribute, an OSH must be provided to handle the `read-attribute` operation.
- Most resources need OSHs created for the `add` and `remove` operations. These are passed to the `Parameters` object given to the `SimpleResourceDefinition` constructor, for use by the `SimpleResourceDefinition` in its implementation of the `registerOperations` method.
- If your resource has custom operations, you will instantiate them to register with a `ManagementResourceRegistration`, typically in an implementation of `ResourceDefinition.registerOperations`
- If an OSH needs to tell the `OperationContext` to add additional steps to do further handling, the OSH will create another OSH to execute that step. This second OSH is typically an inner class of the first OSH.

11.12.7 Operation Execution and the OperationContext

When the `ModelController` at the heart of the WildFly Core management layer handles a request to execute an operation, it instantiates an implementation of the `OperationContext` interface to do the work. The `OperationContext` is configured with an initial list of operation steps it must execute. This is done in one of two ways:

- During boot, multiple steps are configured, one for each operation in the list generated by the parser of the xml configuration file. For each operation, the `ModelController` finds the `ManagementResourceRegistration` that matches the address of the operation and finds the `OperationStepHandler` registered with that MRR for the operation's name. A step is added to the `OperationContext` for each operation by providing the operation `ModelNode` itself, plus the `OperationStepHandler`.
- After boot, any management request involves only a single operation, so only a single step is added. (Note that a `composite` operation is still a single operation; it's just one that internally executes via multiple steps.)

The `ModelController` then asks the `OperationContext` to execute the operation.

The `OperationContext` acts as both the engine for operation execution, and as the interface provided to `OperationStepHandler` implementations to let them interact with the rest of the system.

Execution Process

Operation execution proceeds via execution by the `OperationContext` of a series of "steps" with an `OperationStepHandler` doing the key work for each step. As mentioned above, during boot the OC is initially configured with a number of steps, but post boot operations involve only a single step initially. But even a post-boot operation can end up involving numerous steps before completion. In the case of a `/:read-resource(recursive=true)` operation, thousands of steps might execute. This is possible because one of the key things an `OperationStepHandler` can do is ask the `OperationContext` to add additional steps to execute later.

Execution proceeds via a series of "stages", with a queue of steps maintained for each stage. An `OperationStepHandler` can tell the `OperationContext` to add a step for any stage equal to or later than the currently executing stage. The instruction can either be to add the step to the head of the queue for the stage or to place it at the end of the stage's queue.

Execution of a stage continues until there are no longer any steps in the stage's queue. Then an internal transition task can execute, and the processing of the next stage's steps begins.

Here is some brief information about each stage:

Stage.MODEL

This stage is concerned with interacting with the persistent configuration model, either making changes to it or reading information from it. Handlers for this stage should not make changes to the runtime, and handlers running after this stage should not make changes to the persistent configuration model.

If any step fails during this stage, the operation will automatically roll back. Rollback of MODEL stage failures cannot be turned off. Rollback during boot results in abort of the process start.

The initial step or steps added to the `OperationContext` by the `ModelController` all execute in Stage.MODEL. This means that all `OperationStepHandler` instances your extension registers with a `ManagementResourceRegistration` must be designed for execution in `Stage.MODEL`. If you need work done in later stages your `Stage.MODEL` handler must add a step for that work.

When this stage completes, the `OperationContext` internally performs model validation work before proceeding on to the next stage. Validation failures will result in rollback.

Stage.RUNTIME

This stage is concerned with interacting with the server runtime, either reading from it or modifying it (e.g. installing or removing services or updating their configuration.) By the time this stage begins, all model changes are complete and model validity has been checked. So typically handlers in this stage read their inputs from the model, not from the original `operation` `ModelNode` provided by the user.

Most `OperationStepHandler` logic written by extension authors will be for Stage.RUNTIME. The vast majority of Stage.MODEL handling can best be performed by the base handler classes WildFly Core provides in its `controller` module. (See below for more on those.)

During boot failures in `Stage.RUNTIME` will not trigger rollback and abort of the server boot. After boot, by default failures here will trigger rollback, but users can prevent that by using the `rollback-on-runtime-failure` header. However, a RuntimeException thrown by a handler will trigger rollback.

At the end of `Stage.RUNTIME`, the `OperationContext` blocks waiting for the MSC service container to stabilize (i.e. for all services to have reached a rest state) before moving on to the next stage.

Stage.VERIFY

Service container verification work is performed in this stage, checking that any MSC changes made in `Stage.RUNTIME` had the expected effect. Typically extension authors do not add any steps in this stage, as the steps automatically added by the `OperationContext` itself are all that are needed. You can add a step here though if you have an unusual use case where you need to verify something after MSC has stabilized.

Handlers in this stage should not make any further runtime changes; their purpose is simply to do verification work and fail the operation if verification is unsuccessful.

During boot failures in `Stage.VERIFY` will not trigger rollback and abort of the server boot. After boot, by default failures here will trigger rollback, but users can prevent that by using the `rollback-on-runtime-failure` header. However, a RuntimeException thrown by a handler will trigger rollback.

There is no special transition work at the end of this stage.

Stage.DOMAIN

Extension authors should not add steps in this stage; it is only for use by the kernel.

Steps needed to execute rollout across the domain of an operation that affects multiple processes in a managed domain run here. This stage is only run on Host Contoller processes, never on servers.

Stage.DONE and ResultHandler / RollbackHandler Execution

This stage doesn't maintain a queue of steps; no `OperationStepHandler` executes here. What does happen here is persistence of any configuration changes to the xml file and commit or rollback of changes affecting multiple processes in a managed domain.

While no `OperationStepHandler` executes in this stage, following persistence and transaction commit all `ResultHandler` or `RollbackHandler` callbacks registered with the `OperationContext` by the steps that executed are invoked. This is done in the reverse order of step execution, so the callback for the last step to run is the first to be executed. The most common thing for a callback to do is to respond to a rollback by doing whatever is necessary to reverse changes made in `Stage.RUNTIME`. (No reversal of `Stage.MODEL` changes is needed, because if an operation rolls back the updated model produced by the operation is simply never published and is discarded.)

Tips About Adding Steps

Here are some useful tips about how to add steps:

- Add a step to the head of the current stage's queue if you want it to execute next, prior to any other steps. Typically you would use this technique if you are trying to decompose some complex work into pieces, with reusable logic handling each piece. There would be an `OperationStepHandler` for each part of the work, added to the head of the queue in the correct sequence. This would be a pretty advanced use case for an extension author but is quite common in the handlers provided by the kernel.
- Add a step to the end of the queue if either you don't care when it executes or if you do care and want to be sure it executes after any already registered steps.
    - A very common example of this is a `Stage.MODEL` handler adding a step for its associated `Stage.RUNTIME` work. If there are multiple model steps that will execute (e.g. at boot or as part of handling a `composite`), each will want to add a runtime step, and likely the best order for those runtime steps is the same as the order of the model steps. So if each adds its runtime step at the end, the desired result will be achieved.
    - A more sophisticated but important scenario is when a step may or may not be executing as part of a larger set of steps, i.e. it may be one step in a `composite` or it may not. There is no way for the handler to know. But it can assume that if it is part of a composite, the steps for the other operations in the composite **are already registered in the queue**. (The handler for the `composite` op guarantees this.) So, if it wants to do some work (say validation of the relationship between different attributes or resources) the input to which may be affected by possible other already registered steps, instead of doing that work itself, it should register a different step at the **end** of the queue and have that step do the work. This will ensure that when the validation step runs, the other steps in the `composite` will have had a chance to do their work. **Rule of thumb: always doing any extra validation work in an added step.**

Passing Data to an Added Step

Often a handler author will want to share state between the handler for a step it adds and the handler that added it. There are a number of ways this can be done:

- Very often the `OperationStepHandler` for the added class is an inner class of the handler that adds it. So here sharing state is easily done using final variables in the outer class.
- The handler for the added step can accept values passed to its constructor which can serve as shared state.
- The `OperationContext` includes an Attachment API which allows arbitary data to be attached to the context and retrieved by any handler that has access to the attachment key.
- The `OperationContext.addStep` methods include overloaded variants where the caller can pass in an `operation` `ModelNode` that will in turn be passed to the `execute` method of the handler for the added step. So, state can be passed via this `ModelNode`. It's important to remember though that the `address` field of the `operation` will govern what the `OperationContext` sees as the target of operation when that added step's handler executes.

Controlling Output from an Added Step

When an `OperationStepHandler` wants to report an operation result, it calls the `OperationContext.getResult()` method and manipulates the returned `ModelNode`. Similarly for failure messages it can call `OperationContext.getFailureDescription()`. The usual assumption when such a call is made is that the result or failure description being modified is the one at the root of the response to the end user. But this is not necessarily the case.

When an `OperationStepHandler` adds a step it can use one of the overloaded `OperationContext.addStep` variants that takes a `response` `ModelNode` parameter. If it does, whatever `ModelNode` it passes in will be what is updated as a result of `OperationContext.getResult()` and `OperationContext.getFailureDescription()` calls by the step's handler. This node does not need to be one that is directly associated with the response to the user.

How then does the handler that adds a step in this manner make use of whatever results the added step produces, since the added step will not run until the adding step completes execution? There are a couple of ways this can be done.

The first is to add yet another step, and provide it a reference to the `response` node used by the second step. It will execute after the second step and can read its response and use it in formulating its own response.

The second way involves using a `ResultHandler`. The `ResultHandler` for a step will execute **after** any step that it adds executes. And, it is legal for a `ResultHandler` to manipulate the "result" value for an operation, or its "failure-description" in case of failure. So, the handler that adds a step can provide to its `ResultHandler` a reference to the `response` node it passed to `addStep`, and the `ResultHandler` can in turn and use its contents to manipulate its own response.

This kind of handling wouldn't commonly be done by extension authors and great care needs to be taken if it is done. It is often done in some of the kernel handlers.

OperationStepHandler use of the OperationContext

All useful work an `OperationStepHandler` performs is done by invoking methods on the `OperationContext`. The `OperationContext` interface is extensively javadoced, so this section will just provide a brief partial overview. The OSH can use the `OperationContext` to:

- Learn about the environment in which it is executing (`getProcessType`, `getRunningMode`, `isBooting`, `getCurrentStage`, `getCallEnvironment`, `getSecurityIdentity`, `isDefaultRequiresRuntime`, `isNormalServer`)
- Learn about the operation (`getCurrentAddress`, `getCurrentAddressValue`, `getAttachmentStream`, `getAttachmentStreamCount`)
- Read the `Resource` tree (`readResource`, `readResourceFromRoot`, `getOriginalRootResource`)
- Manipulate the `Resource` tree (`createResource`, `addResource`, `readResourceForUpdate`, `removeResource`)
- Read the resource type information (`getResourceRegistration`, `getRootResourceRegistration`)
- Manipulate the resource type information (`getResourceRegistrationForUpdate`)
- Read the MSC service container (`getServiceRegistry(false)`)
- Manipulate the MSC service container (`getServiceTarget`, `getServiceRegistry(true)`, `removeService`)
- Manipulate the process state (`reloadRequired`, `revertReloadRequired`, `restartRequired`, `revertRestartRequired`
- Resolve expressions (`resolveExpressions`)
- Manipulate the operation response (`getResult`, `getFailureDescription`, `attachResultStream`, `runtimeUpdateSkipped`)
- Force operation rollback (`setRollbackOnly`)
- Add other steps (`addStep`)
- Share data with other steps (`attach`, `attachIfAbsent`, `getAttachment`, `detach`)
- Work with capabilities (numerous methods)
- Emit notifications (`emit`)
- Request a callback to a `ResultHandler` or `RollbackHandler` (`completeStep`)

Locking and Change Visibility

The `ModelController` and `OperationContext` work together to ensure that only one operation at a time is modifying the state of the system. This is done via an exclusive lock maintained by the `ModelController`. Any operation that does not need to write never requests the lock and is able to proceed without being blocked by an operation that holds the lock (i.e. writes do not block reads.) If two operations wish to concurrently write, one or the other will get the lock and the loser will block waiting for the winner to complete and release the lock.

The `OperationContext` requests the exclusive lock the first time any of the following occur:

- A step calls one of its methods that indicates a wish to modify the resource tree (`createResource`, `addResource`, `readResourceForUpdate`, `removeResource`)
- A step calls one of its methods that indicates a wish to modify the `ManagementResourceRegistration` tree (`getResourceRegistrationForUpdate`)
- A step calls one of its methods that indicates a desire to change MSC services (`getServiceTarget`, `removeService` or `getServiceRegistry` with the `modify` param set to `true`)
- A step calls one of its methods that manipulates the capability registry (various)
- A step explicitly requests the lock by calling the `acquireControllerLock` method (doing this is discouraged)

The step that acquired the lock is tracked, and the lock is released when the `ResultHandler` added by that step has executed. (If the step doesn't add a result handler, a default no-op one is automatically added).

When an operation first expresses a desire to manipulate the `Resource` tree or the capability registry, a private copy of the tree or registry is created and thereafter the `OperationContext` works with that copy. The copy is published back to the `ModelController` in `Stage.DONE` if the operation commits. Until that happens any changes to the tree or capability registry made by the operation are invisible to other threads. If the operation does not commit, the private copies are simply discarded.

However, the `OperationContext` does not make a private copy of the `ManagementResourceRegistration` tree before manipulating it, nor is there a private copy of the MSC service container. So, any changes made by an operation to either of those are immediately visible to other threads.

11.12.8 Resource Interface

An instance of the `Resource` interface holds the state for a particular instance of a type defined by a `ManagementResourceRegistration`. Referring back to the analogy mentioned earlier the `ManagementResourceRegistration` is analogous to a Java class while the `Resource` is analogous to an instance of that class.

The `Resource` makes available state information, primarily

- Some descriptive metadata, such as its address, whether it is runtime-only and whether it represents a proxy to a another primary resource that resides on another process in a managed domain
- A `ModelNode` of `ModelType.OBJECT` whose keys are the resource's attributes and whose values are the attribute values
- Links to child resources such that the resources form a tree

Creating Resources

Typically extensions create resources via `OperationStepHandler` calls to the `OperationContext.createResource` method. However it is allowed for handlers to use their own `Resource` implementations by instantiating the resource and invoking `OperationContext.addResource`. The `AbstractModelResource` class can be used as a base class.

Runtime-Only and Synthetic Resources and the PlaceholderResourceEntry Class

A runtime-only resource is one whose state is not persisted to the xml configuration file. Many runtime-only resources are also "synthetic" meaning they are not added or removed as a result of user initiated management operations. Rather these resources are "synthesized" in order to allow users to use the management API to examine some aspect of the internal state of the process. A good example of synthetic resources are the resources in the `/core-service=platform-mbeans` branch of the resource tree. There are resources there that represent various aspects of the JVM (classloaders, memory pools, etc) but which resources are present entirely depends on what the JVM is doing, not on any management action. Another example are resources representing "core queues" in the WildFly messaging and messaging-artemismq subsystems. Queues are created as a result of activity in the message broker which may not involve calls to the management API. But for each such queue a management resource is available to allow management users to perform management operations against the queue.

It is a requirement of execution of a management operation that the `OperationContext` can navigate through the resource tree to a `Resource` object located at the address specified. This requirement holds true even for synthetic resources. How can this be handled, given the fact these resources are not created in response to management operations?

The trick involves using special implementations of `Resource`. Let's imagine a simple case where we have a parent resource which is fairly normal (i.e. it holds persistent configuration and is added via a user's `add` operation) except for the fact that one of its child types represents synthetic resources (e.g. message queues). How would this be handled?

First, the parent resource would require a custom implementation of the `Resource` interface. The `OperationStepHandler` for the `add` operation would instantiate it, providing it with access to whatever API is needed for it to work out what items exist for which a synthetic resource should be made available (e.g. an API provided by the message broker that provides access to its queues). The `add` handler would use the `OperationContext.addResource` method to tie this custom resource into the overall resource tree.

The custom `Resource` implementation would use special implementations of the various methods that relate to accessing children. For all calls that relate to the synthetic child type (e.g. core-queue) the custom implementation would use whatever API call is needed to provide the correct data for that child type (e.g. ask the message broker for the names of queues).

A nice strategy for creating such a custom resource is to use delegation. Use `Resource.Factory.create}()` to create a standard resource. Then pass it to the constructor of your custom resource type for use as a delegate. The custom resource type's logic is focused on the synthetic children; all other work it passes on to the delegate.

What about the synthetic resources themselves, i.e. the leaf nodes in this part of the tree? These are created on the fly by the parent resource in response to `getChild`, `requireChild`, `getChildren` and `navigate` calls that target the synthetic resource type. These created-on-the-fly resources can be very lightweight, since they store no configuration model and have no children. The `PlaceholderResourceEntry` class is perfect for this. It's a very lightweight `Resource` implementation with minimal logic that only stores the final element of the resource's address as state.

See `LoggingResource` in the WildFly Core logging subsystem for an example of this kind of thing. Searching for other uses of `PlaceholderResourceEntry` will show other examples.

11.12.9 DeploymentUnitProcessor Interface

TODO

11.12.10 Useful classes for implementing OperationStepHandler

The WildFly Core `controller` module includes a number of `OperationStepHandler` implementations that in some cases you can use directly, and that in other cases can serve as the base class for your own handler implementation. In all of these a general goal is to eliminate the need for your code to do anything in `Stage.MODEL` while providing support for whatever is appropriate for `Stage.RUNTIME`.

Add Handlers

`AbstractAddStepHandler` is a base class for handlers for `add` operations. There are a number of ways you can configure its behavior, the most commonly used of which are to:

- Configure its behavior in `Stage.MODEL` by passing to its constructor `AttributeDefinition` and `RuntimeCapability` instances for the attributes and capabilities provided by the resource. The handler will automatically validate the operation parameters whose names match the provided attributes and store their values in the model of the newly added `Resource`. It will also record the presence of the given capabilities.
- Control whether a `Stage.RUNTIME` step for the operation needs to be added, by overriding the `protected boolean requiresRuntime(OperationContext context)` method. Doing this is atypical; the standard behavior in the base class is appropriate for most cases.
- Implement the primary logic of the `Stage.RUNTIME` step by overriding the `protected void performRuntime(final OperationContext context, final ModelNode operation, final Resource resource)` method. This is typically the bulk of the code in an `AbstractAddStepHandler` subclass. This is where you read data from the `Resource` model and use it to do things like configure and install MSC services.
- Handle any unusual needs of any rollback of the `Stage.RUNTIME` step by overriding `protected void rollbackRuntime(OperationContext context, final ModelNode operation, final Resource resource)`. Doing this is not typically needed, since if the rollback behavior needed is simply to remove any MSC services installed in `performRuntime`, the `OperationContext` will do this for you automatically.

`AbstractBoottimeAddStepHandler` is a subclass of `AbstractAddStepHandler` meant for use by `add` operations that should only do their normal `Stage.RUNTIME` work in server, boot, with the server being put in `reload-required` if executed later. Primarily this is used for `add` operations that register `DeploymentUnitProcessor` implementations, as this can only be done at boot.

Usage of `AbstractBoottimeAddStepHandler` is the same as for `AbstractAddStepHandler` except that instead of overriding `performRuntime` you override `protected void performBoottime(OperationContext context, ModelNode operation, Resource resource)`.

A typical thing to do in `performBoottime` is to add a special step that registers one or more `DeploymentUnitProcessor`s.

```
@Override
    public void performBoottime(OperationContext context, ModelNode operation, final Resource
resource)
            throws OperationFailedException {

        context.addStep(new AbstractDeploymentChainStep() {
            @Override
            protected void execute(DeploymentProcessorTarget processorTarget) {


processorTarget.addDeploymentProcessor(RequestControllerExtension.SUBSYSTEM_NAME,
Phase.STRUCTURE, Phase.STRUCTURE_GLOBAL_REQUEST_CONTROLLER, new
RequestControllerDeploymentUnitProcessor());
            }
        }, OperationContext.Stage.RUNTIME);

        ... do other things
```

Remove Handlers

TODO `AbstractRemoveStepHandler` `ServiceRemoveStepHandler`

Write attribute handlers

TODO `AbstractWriteAttributeHandler`

Reload-required handlers

`ReloadRequiredAddStepHandler` `ReloadRequiredRemoveStepHandler` `ReloadRequiredWriteAttributeHandler`

Use these for cases where, post-boot, the change to the configuration model made by the operation cannot be reflected in the runtime until the process is reloaded. These handle the mechanics of recording the need for reload and reverting it if the operation rolls back.

Restart Parent Resource Handlers

`RestartParentResourceAddHandler` `RestartParentResourceRemoveHandler` `RestartParentWriteAttributeHandler`

Use these in cases where a management resource doesn't directly control any runtime services, but instead simply represents a chunk of configuration that a parent resource uses to configure services it installs. (Really, this kind of situation is now considered to be a poor management API design and is discouraged. Instead of using child resources for configuration chunks, complex attributes on the parent resource should be used.)

These handlers help you deal with the mechanics of the fact that, post-boot, any change to the child resource likely requires a restart of the service provided by the parent.

Model Only Handlers

`ModelOnlyAddStepHandler` `ModelOnlyRemoveStepHandler` `ModelOnlyWriteAttributeHandler`

Use these for cases where the operation never affects the runtime, even at boot. All it does is update the configuration model. In most cases such a thing would be odd. These are primarily useful for legacy subsystems that are no longer usable on current version servers and thus will never do anything in the runtime. However, current version Domain Controllers must be able to understand the subsystem's configuration model to allow them to manage older Host Controllers running previous versions where the subsystem is still usable by servers. So these handlers allow the DC to maintain the configuration model for the subsystem.

Misc

`AbstractRuntimeOnlyHandler` is used for custom operations that don't involve the configuration model. Create a subclass and implement the `protected abstract void executeRuntimeStep(OperationContext context, ModelNode operation)` method. The superclass takes care of adding a `Stage.RUNTIME` step that calls your method.

`ReadResourceNameOperationStepHandler` is for cases where a resource type includes a 'name' attribute whose value is simply the value of the last element in the resource's address. There is no need to store the value of such an attribute in the resource's model, since it can always be determined from the resource address. But, if the value is not stored in the resource model, when the attribute is registered with `ManagementResourceRegistration.registerReadAttribute` an `OperationStepHandler` to handle the `read-attribute` operation must be provided. Use `ReadResourceNameOperationStepHandler` for this. (Note that including such an attribute in your management API is considered to be poor practice as it's just redundant data.)

11.13 WildFly 9 JNDI Implementation

11.13.1 Introduction

This page proposes a reworked WildFly JNDI implementation, and new/updated APIs for WildFly subsystem and EE deployment processors developers to bind new resources easier.

To support discussion in the community, the content includes a big focus on comparing WildFly 8 JNDI implementation with the new proposal, and should later evolve to the prime guide for WildFly developers needing to interact with JNDI at subsystem level.

11.13.2 Architecture

WildFly relies on MSC to provide the data source for the JNDI tree. Each resource bound in JNDI is stored in a MSC service (BinderService), and such services are installed as children of subsystem/deployment services, for an automatically unbound as consequence of uninstall of the parent services.

Since there is the need to know what entries are bound, and MSC does not provides that, there is also the (ServiceBased)NamingStore concept, which internally manage the set of service names bound. There are multiple naming stores in every WildFly instance, serving different JNDI namespaces:

- java:comp - the standard EE namespace for entries scoped to a specific component, such as an EJB
- java:module - the standard EE namespace for entries scoped to specific module, such as an EJB jar, and shared by all components in it
- java:app - the standard EE namespace for entries scoped to a specific application, i.e. EAR, and shared by all modules in it
- java:global - the standard EE namespace for entries shared by all deployments
- java:jboss - a proprietary namespace "global" namespace
- java:jboss/exported - a proprietary "global" namespace which entries are exposed to remote JNDI
- java: - any entries not in the other namespaces

One particular implementation choice, to save resources, is that JNDI contexts by default are not bound, the naming stores will search for any entry bound with a name that is a child of the context name, if found then its assumed the context exists.

The reworked implementation introduces shared/global java:comp, java:module and java:app namespaces. Any entry bound on these will automatically be available to every EE deployment scoped instance of these namespaces, what should result in a significant reduction of binder services, and also of EE deployment processors. Also, the Naming subsystem may now configure bind on these shared contexts, and these contexts will be available when there is no EE component in the invocation, which means that entries such as java:comp/DefaultDatasource will always be available.

11.13.3 Binding APIs

WildFly Naming subsystem exposes high level APIs to bind new JNDI resources, there is no need to deal with the low level BinderService type anymore.

Subsystem

At the lowest level a JNDI entry is bound by installing a BinderService to a ServiceTarget:

```
   /**
     * Binds a new entry to JNDI.
     * @param serviceTarget the binder service's target
     * @param name the new JNDI entry's name
     * @param value the new JNDI entry's value
     */
    private ServiceController<?> bind(ServiceTarget serviceTarget, String name, Object value) {

 // the bind info object provides MSC service names to use when creating the binder service
        final ContextNames.BindInfo bindInfo = ContextNames.bindInfoFor(name);
        final BinderService binderService = new BinderService(bindInfo.getBindName());

 // the entry's value is provided by a managed reference factory,
        // since the value may need to be obtained on lookup (e.g. EJB reference)
        final ManagedReferenceFactory managedReferenceFactory = new
ImmediateManagedReferenceFactory(value);

 return serviceTarget
                // add binder service to specified target
                .addService(bindInfo.getBinderServiceName(), binderService)
                // when started the service will be injected with the factory
                .addInjection(binderService.getManagedObjectInjector(), managedReferenceFactory)
                // the binder service depends on the related naming store service,
                // and on start/stop will add/remove its service name
                .addDependency(bindInfo.getParentContextServiceName(),
                        ServiceBasedNamingStore.class,
                        binderService.getNamingStoreInjector())
                .install();
    }
```

But the example above is the simplest usage possible, it may become quite complicated if the entry's value is not immediately available, for instance it is a value in another MSC service, or is a value in another JNDI entry. It's also quite easy to introduce bugs when working with the service names, or incorrectly assume that other MSC functionality, such as alias names, may be used.

Using the new high level API, it's as simple as:

```
// bind an immediate value
ContextNames.bindInfoFor("java:comp/ORB").bind(serviceTarget, this.orb);


// bind value from another JNDI entry (an alias/linkref)
ContextNames.bindInfoFor(“java:global/x").bind(serviceTarget, new JndiName(“java:jboss/x"));


// bind value obtained from a MSC service
ContextNames.bindInfoFor(“java:global/z").bind(serviceTarget, serviceName);
```

If there is the need to access the binder's service builder, perhaps to add a service verification handler or simply not install the binder service right away:

```
ContextNames.bindInfoFor("java:comp/ORB").builder(serviceTarget, verificationHandler,
ServiceController.Mode.ON_DEMAND).installService(this.orb);
```

EE Deployment

With respect to EE deployments, the subsystem API should not be used, since bindings may need to be discarded/overridden, thus a EE deployment processor should add a new binding in the form of a BindingConfiguration, to the EeModuleDescription or ComponentDescription, depending if the bind is specific to a component or not. An example of a deployment processor adding a binding:

```
public class ModuleNameBindingProcessor implements DeploymentUnitProcessor {

    // jndi name objects are immutable
    private static final JndiName JNDI_NAME_java_module_ModuleName = new
JndiName("java:module/ModuleName");

    @Override
    public void deploy(DeploymentPhaseContext phaseContext) throws
DeploymentUnitProcessingException {

 final DeploymentUnit deploymentUnit = phaseContext.getDeploymentUnit();
        // skip deployment unit if it's the top level EAR
        if (DeploymentTypeMarker.isType(DeploymentType.EAR, deploymentUnit)) {
            return;
        }

 // the module's description is in the DUs attachments
        final EEModuleDescription moduleDescription = deploymentUnit
                .getAttachment(org.jboss.as.ee.component.Attachments.EE_MODULE_DESCRIPTION);
        if (moduleDescription == null) {
            return;
        }

 // add the java:module/ModuleName binding
        // the value's injection source for an immediate available value
        final InjectionSource injectionSource = new
ImmediateInjectionSource(moduleDescription.getModuleName());

 // add the binding configuration to the module's description bindings configurations
        moduleDescription.getBindingConfigurations()
                .addDeploymentBinding(new BindingConfiguration(JNDI_NAME_java_module_ModuleName,
injectionSource));
    }

    //...
}
```

When adding the binding configuration use:

- addDeploymentBinding() for a binding that may not be overriden, such as the ones found in xml descriptors
- addPlatformBinding() for a binding which may be overriden by a deployment descriptor bind or annotation, for instance java:comp/DefaultDatasource

A deployment processor may now also add a binding configuration to all components in a module:

```
moduleDescription.getBindingConfigurations().addPlatformBindingToAllComponents(bindingConfiguratio
```

In the reworked implementation there is now no need to behave differently considering the deployment type, for instance if deployment is a WAR or app client, the Module/Component BindingConfigurations objects handle all of that. The processor should simply go for the 3 use cases: module binding, component binding or binding shared by all components.

All deployment binding configurations MUST be added before INSTALL phase, this is needed because on such phase, when the bindings are actually done, there must be a final set of deployment binding names known, such information is need to understand if a resource injection targets entries in the global or scoped EE namespaces.

Most cases for adding bindings to EE deployments are in the context of a processor deploying a XML descriptor, or scanning deployment classes for annotations, and there abstract types, such as the AbstractDeploymentDescriptorBindingsProcessor, which simplifies greatly the processor code for such use cases.

One particular use case is the parsing of EE Resource Definitions, and the reworked implementation provides high level abstract deployment processors for both XML descriptor and annotations, an example for each:

```
/**
 * Deployment processor responsible for processing administered-object deployment descriptor
elements
 *
 * @author Eduardo Martins
 */
public class AdministeredObjectDefinitionDescriptorProcessor extends
ResourceDefinitionDescriptorProcessor {

    @Override
    protected void processEnvironment(RemoteEnvironment environment,
ResourceDefinitionInjectionSources injectionSources) throws DeploymentUnitProcessingException {
        final AdministeredObjectsMetaData metaDatas = environment.getAdministeredObjects();
        if (metaDatas != null) {
            for(AdministeredObjectMetaData metaData : metaDatas) {

injectionSources.addResourceDefinitionInjectionSource(getResourceDefinitionInjectionSource(metaDat
}
        }
    }

    private ResourceDefinitionInjectionSource getResourceDefinitionInjectionSource(final
AdministeredObjectMetaData metaData) {
        final String name = metaData.getName();
        final String className = metaData.getClassName();
        final String resourceAdapter = metaData.getResourceAdapter();
        final AdministeredObjectDefinitionInjectionSource resourceDefinitionInjectionSource =
new AdministeredObjectDefinitionInjectionSource(name, className, resourceAdapter);
        resourceDefinitionInjectionSource.setInterface(metaData.getInterfaceName());
        if (metaData.getDescriptions() != null) {

resourceDefinitionInjectionSource.setDescription(metaData.getDescriptions().toString());
        }
        resourceDefinitionInjectionSource.addProperties(metaData.getProperties());
        return resourceDefinitionInjectionSource;
    }

}
```

and

```
/**
 * Deployment processor responsible for processing {@link
javax.resource.AdministeredObjectDefinition} and {@link
javax.resource.AdministeredObjectDefinitions}.
 *
 * @author Jesper Pedersen
 * @author Eduardo Martins
 */
public class AdministeredObjectDefinitionAnnotationProcessor extends
ResourceDefinitionAnnotationProcessor {

    private static final DotName ANNOTATION_NAME =
DotName.createSimple(AdministeredObjectDefinition.class.getName());
    private static final DotName COLLECTION_ANNOTATION_NAME =
DotName.createSimple(AdministeredObjectDefinitions.class.getName());

    @Override
    protected DotName getAnnotationDotName() {
        return ANNOTATION_NAME;
    }

    @Override
    protected DotName getAnnotationCollectionDotName() {
        return COLLECTION_ANNOTATION_NAME;
    }

    @Override
    protected ResourceDefinitionInjectionSource processAnnotation(AnnotationInstance
annotationInstance) throws DeploymentUnitProcessingException {
        final String name = AnnotationElement.asRequiredString(annotationInstance,
AnnotationElement.NAME);
        final String className = AnnotationElement.asRequiredString(annotationInstance,
"className");
        final String ra = AnnotationElement.asRequiredString(annotationInstance,
"resourceAdapter");
        final AdministeredObjectDefinitionInjectionSource
directAdministeredObjectInjectionSource =
                new AdministeredObjectDefinitionInjectionSource(name, className, ra);

directAdministeredObjectInjectionSource.setDescription(AnnotationElement.asOptionalString(annotati
AdministeredObjectDefinitionInjectionSource.DESCRIPTION));

directAdministeredObjectInjectionSource.setInterface(AnnotationElement.asOptionalString(annotation
AdministeredObjectDefinitionInjectionSource.INTERFACE));

directAdministeredObjectInjectionSource.addProperties(AnnotationElement.asOptionalStringArray(anno
AdministeredObjectDefinitionInjectionSource.PROPERTIES));
        return directAdministeredObjectInjectionSource;
    }

}
```

The abstract processors with respect to Resource Definitions are already submitted through WFLY-3292's PR.

11.13.4 Resource Ref Processing

TODO for now no changes on this in the reworked WildFly Naming.

11.14 Working with WildFly Capabilities

An extension to WildFly will likely want to make use of services provided by the WildFly kernel, may want to make use of services provided by other subsystems, and may wish to make functionality available to other extensions. Each of these cases involves integration between different parts of the system. In releases prior to WildFly 10, this kind of integration was done on an ad-hoc basis, resulting in overly tight coupling between different parts of the system and overly weak integration contracts. For example, a service installed by subsystem A might depend on a service installed by subsystem B, and to record that dependency A's authors copy a ServiceName from B's code, or even refer to a constant or static method from B's code. The result is B's code cannot evolve without risking breaking A. And the authors of B may not even intend for other subsystems to use its services. There is no proper integration contract between the two subsystems.

Beginning with WildFly Core 2 and WildFly 10 the WildFly kernel's management layer provides a mechanism for allowing different parts of the system to integrate with each other in a loosely coupled manner. This is done via WildFly Capabilities. Use of capabilities provides the following benefits:

1. A standard way for system components to define integration contracts for their use by other system components.
2. A standard way for system components to access integration contracts provided by other system components.
3. A mechanism for configuration model referential integrity checking, such that if one component's configuration has an attribute that refers to an other component (e.g. a `socket-binding` attribute in a subsystem that opens a socket referring to that socket's configuration), the validity of that reference can be checked when validating the configuration model.

11.14.1 Capabilities

A capability is a piece of functionality used in a WildFly Core based process that is exposed via the WildFly Core management layer. Capabilities may depend on other capabilities, and this interaction between capabilities is mediated by the WildFly Core management layer.

Some capabilities are automatically part of a WildFly Core based process, but in most cases the configuration provided by the end user (i.e. in standalone.xml, domain.xml and host.xml) determines what capabilities are present at runtime. It is the responsibility of the handlers for management operations to register capabilities and to register any requirements those capabilities may have for the presence of other capabilities. This registration is done during the MODEL stage of operation execution

A capability has the following basic characteristics:

1. It has a name.
2. It may install an MSC service that can be depended upon by services installed by other capabilities. If it does, it provides a mechanism for discovering the name of that service.
3. It may expose some other API not based on service dependencies allowing other capabilities to integrate with it at runtime.
4. It may depend on, or **require** other capabilities.

During boot of the process, and thereafter whenever a management operation makes a change to the process' configuration, at the end of the MODEL stage of operation execution the kernel management layer will validate that all capabilities required by other capabilities are present, and will fail any management operation step that introduced an unresolvable requirement. This will be done before execution of the management operation proceeds to the RUNTIME stage, where interaction with the process' MSC Service Container is done. As a result, in the RUNTIME stage the handler for an operation can safely assume that the runtime services provided by a capability for which it has registered a requirement are available.

Comparison to other concepts

Capabilities vs modules

A JBoss Modules module is the means of making resources available to the classloading system of a WildFly Core based process. To make a capability available, you must package its resources in one or more modules and make them available to the classloading system. But a module is not a capability in and of itself, and simply copying a module to a WildFly installation does not mean a capability is available. Modules can include resources completely unrelated to management capabilities.

Capabilities vs Extensions

An extension is the means by which the WildFly Core management layer is made aware of manageable functionality that is not part of the WildFly Core kernel. The extension registers with the kernel new management resource types and handlers for operations on those resources. One of the things a handler can do is register or unregister a capability and its requirements. An extension may register a single capability, multiple capabilities, or possibly none at all. Further, not all capabilities are registered by extensions; the WildFly Core kernel itself may register a number of different capabilities.

Capability Names

Capability names are simple strings, with the dot character serving as a separator to allow namespacing.

The 'org.wildfly' namespace is reserved for projects associated with the WildFly organization on github (https://github.com/wildfly).

Statically vs Dynamically Named Capabilities

The full name of a capability is either statically known, or it may include a statically known base element and then a dynamic element. The dynamic part of the name is determined at runtime based on the address of the management resource that registers the capability. For example, the management resource at the address '/socket-binding-group=standard-sockets/socket-binding=web' will register a dynamically named capability named 'org.wildlfy.network.socket-binding.web'. The 'org.wildlfy.network.socket-binding' portion is the static part of the name.

All dynamically named capabilities that have the same static portion of their name should provide a consistent feature set and set of requirements.

Service provided by a capability

Typically a capability functions by registering a service with the WildFly process' MSC ServiceContainer, and then dependent capabilities depend on that service. The WildFly Core management layer orchestrates registration of those services and service dependencies by providing a means to discover service names.

Custom integration APIs provided by a capability

Instead of or in addition to providing MSC services, a capability may expose some other API to dependent capabilities. This API must be encapsulated in a single class (although that class can use other non-JRE classes as method parameters or return types).

Capability Requirements

A capability may rely on other capabilities in order to provide its functionality at runtime. The management operation handlers that register capabilities are also required to register their requirements.

There are three basic types of requirements a capability may have:

- Hard requirements. The required capability must always be present for the dependent capability to function.
- Optional requirements. Some aspect of the configuration of the dependent capability controls whether the depended on capability is actually necessary. So the requirement cannot be known until the running configuration is analyzed.
- Runtime-only requirements. The dependent capability will check for the presence of the depended upon capability at runtime, and if present it will utilize it, but if it is not present it will function properly without the capability. There is nothing in the dependent capability's configuration that controls whether the depended on capability must be present. Only capabilities that declare themselves as being suitable for use as a runtime-only requirement should be depended upon in this manner.

Hard and optional requirements may be for either statically named or dynamically named capabilities. Runtime-only requirements can only be for statically named capabilities, as such a requirement cannot be specified via configuration, and without configuration the dynamic part of the required capability name is unknown.

Supporting runtime-only requirements

Not all capabilities are usable as a runtime-only requirement.

Any dynamically named capability is not usable as a runtime-only requirement.

For a capability to support use as a runtime-only requirement, it must guarantee that a configuration change to a running process that removes the capability will not impact currently running capabilities that have a runtime-only requirement for it. This means:

- A capability that supports runtime-only usage must ensure that it never removes its runtime service except via a full process reload.
- A capability that exposes a custom integration API generally is not usable as a runtime-only requirement. If such a capability does support use as a runtime-only requirement, it must ensure that any functionality provided via its integration API remains available as long as a full process reload has not occurred.

11.14.2 Capability Contract

A capability provides a stable contract to users of the capability. The contract includes the following:

- The name of the capability (including whether it is dynamically named).
- Whether it installs an MSC Service, and if it does, the value type of the service. That value type then becomes a stable API users of the capability can rely upon.
- Whether it provides a custom integration API, and if it does, the type that represents that API. That type then becomes a stable API users of the capability can rely upon.
- Whether the capability supports use as a runtime-only requirement.

Developers can learn about available capabilities and the contracts they provide by reading the WildFly *capabilty registry*.

11.14.3 Capability Registry

The WildFly organization on github maintains a git repo where information about available capabilities is published.

https://github.com/wildfly/wildfly-capabilities

Developers can learn about available capabilities and the contracts they provide by reading the WildFly capabilty registry.

The README.md file at the root of that repo explains the how to find out information about the registry.

Developers of new capabilities are **strongly encouraged** to document and register their capability by submitting a pull request to the wildfly-capabilities github repo. This both allows others to learn about your capability and helps prevent capability name collisions. Capabilities that are used in the WildFly or WildFly Core code base itself **must** have a registry entry before the code referencing them will be merged.

External organizations that create capabilities should include an organization-specific namespace as part their capability names to avoid name collisions.

11.14.4 Using Capabilities

Now that all the background information is presented, here are some specifics about how to use WildFly capabilities in your code.

Basics of Using Your Own Capability

Creating your capability

A capability is an instance of the immutable `org.jboss.as.controller.capability.RuntimeCapability` class. A capability is usually registered by a resource, so the usual way to use one is to store it in constant in the resource's `ResourceDefinition`. Use a `RuntimeCapability.Builder` to create one.

```
class MyResourceDefinition extends SimpleResourceDefinition {

    static final RuntimeCapability<Void> FOO_CAPABILITY =
RuntimeCapability.Builder.of("com.example.foo").build();

    . . .
}
```

That creates a statically named capability named `com.example.foo`.

If the capability is dynamically named, add the `dynamic` parameter to state this:

```
static final RuntimeCapability<Void> FOO_CAPABILITY =
            RuntimeCapability.Builder.of("com.example.foo", true).build();
```

Most capabilities install a service that requiring capabilities can depend on. If your capability does this, you need to declare the service's *value type* (the type of the object returned by `org.jboss.msc.Service.getValue()`). For example, if FOO_CAPABILITY provides a `Service<javax.sql.DataSource>`:

```
static final RuntimeCapability<Void> FOO_CAPABILITY =
            RuntimeCapability.Builder.of("com.example.foo", DataSource.class).build();
```

For a dynamic capability:

```
static final RuntimeCapability<Void> FOO_CAPABILITY =
           RuntimeCapability.Builder.of("com.example.foo", true, DataSource.class).build();
```

If the capability provides a custom integration API, you need to instantiate an instance of that API:

```
public class JTSCapability {

    static final JTSCapability INSTANCE = new JTSCapability();

    private JTSCapability() {}

    /**
     * Gets the names of the {@link org.omg.PortableInterceptor.ORBInitializer} implementations
that should be included
     * as part of the {@link org.omg.CORBA.ORB#init(String[], java.util.Properties)
initialization of an ORB}.
     *
     * @return the names of the classes implementing {@code ORBInitializer}. Will not be {@code
null}.
     */
    public List<String> getORBInitializerClasses() {
        return Collections.unmodifiableList(Arrays.asList(

"com.arjuna.ats.jts.orbspecific.jacorb.interceptors.interposition.InterpositionORBInitializerImpl"
"com.arjuna.ats.jbossatx.jts.InboundTransactionCurrentInitializer"));
    }
}
```

and provide it to the builder:

```
static final RuntimeCapability<JTSCapability> FOO_CAPABILITY =
            RuntimeCapability.Builder.of("com.example.foo", JTSCapability.INSTANCE).build();
```

For a dynamic capability:

```
static final RuntimeCapability<JTSCapability> FOO_CAPABILITY =
RuntimeCapability.Builder.of("com.example.foo", true, JTSCapability.INSTANCE).build();
```

A capability can provide both a custom integration API and install a service:

```
static final RuntimeCapability<JTSCapability> FOO_CAPABILITY =
            RuntimeCapability.Builder.of("com.example.foo", JTSCapability.INSTANCE)
                .setServiceType(DataSource.class)
                .build();
```

Registering and unregistering your capability

Once you have your capability, you need to ensure it gets registered with the WildFly Core kernel when your resource is added. This is easily done simply by providing a reference to the capability to the resource's `ResourceDefinition`. This assumes your resource definition is a subclass of the standard `org.jboss.as.controller.SimpleResourceDefinition`. `SimpleResourceDefinition` provides a `Parameters` class that provides a builder-style API for setting up all the data needed by your definition. This includes a `setCapabilities` method that can be used to declare the capabilities provided by resources of this type.

```
class MyResourceDefinition extends SimpleResourceDefinition {

    . . .

    MyResourceDefinition() {
        super(new SimpleResourceDefinition.Parameters(PATH, RESOLVER)
            .setAddHandler(MyAddHandler.INSTANCE)
            .setRemoveHandler(MyRemoveHandler.INSTANCE)
            .setCapabilities(FOO_CAPABILITY)
            );
    }
}
```

Your add handler needs to extend the standard `org.jboss.as.controller.AbstractAddStepHandler` class or one of its subclasses:

```
class MyAddHandler extends AbstractAddStepHandler() {
```

`AbstractAddStepHandler`'s logic will register the capability when it executes.

Your remove handler must also extend of the standard `org.jboss.as.controller.AbstractRemoveStepHandler` or one of its subclasses.

```
class MyRemoveHandler extends AbstractRemoveStepHandler() {
```

`AbstractRemoveStepHandler`'s logic will deregister the capability when it executes.

If for some reason you cannot base your `ResourceDefinition` on `SimpleResourceDefinition` or your handlers on `AbstractAddStepHandler` and `AbstractRemoveStepHandler` then you will need to take responsibility for registering the capability yourself. This is not expected to be a common situation. See the implementation of those classes to see how to do it.

Installing, accessing and removing the service provided by your capability

If your capability installs a service, you should use the `RuntimeCapability` when you need to determine the service's name. For example in the `Stage.RUNTIME` handling of your "add" step handler. Here's an example for a statically named capability:

```
class MyAddHandler extends AbstractAddStepHandler() {

    . . .

    @Override
    protected void performRuntime(final OperationContext context, final ModelNode operation,
                                  final Resource resource) throws OperationFailedException {

        ServiceName serviceName = FOO_CAPABILITY.getCapabilityServiceName();
        Service<DataSource> service = createDataSourceService(context, resource);
        context.getServiceTarget().addService(serviceName, service).install();

    }
```

If the capability is dynamically named, get the dynamic part of the name from the `OperationContext` and use that when getting the service name:

```
class MyAddHandler extends AbstractAddStepHandler() {

    . . .

    @Override
    protected void performRuntime(final OperationContext context, final ModelNode operation,
                                  final Resource resource) throws OperationFailedException {

        String myName = context.getCurrentAddressValue();
        ServiceName serviceName = FOO_CAPABILITY.getCapabilityServiceName(myName);
        Service<DataSource> service = createDataSourceService(context, resource);
        context.getServiceTarget().addService(serviceName, service).install();

    }
```

The same patterns should be used when accessing or removing the service in handlers for `remove`, `write-attribute` and custom operations.

If you use `ServiceRemoveStepHandler` for the `remove` operation, simply provide your `RuntimeCapability` to the `ServiceRemoveStepHandler` constructor and it will automatically remove your capability's service when it executes.

Basics of Using Other Capabilities

When a capability needs another capability, it only refers to it by its string name. A capability should not reference the `RuntimeCapability` object of another capability.

Before a capability can look up the service name for a required capability's service, or access its custom integration API, it must first register a requirement for the capability. This must be done in Stage.MODEL, while service name lookups and accessing the custom integration API is done in Stage.RUNTIME.

Registering a requirement for a capability is simple.

Registering a hard requirement for a static capability

If your capability has a hard requirement for a statically named capability, simply declare that to the builder for your `RuntimeCapability`. For example, WildFly's JTS capability requires both a basic transaction support capability and IIOP capabilities:

```
static final RuntimeCapability<JTSCapability> JTS_CAPABILITY =
            RuntimeCapability.Builder.of("org.wildfly.transactions.jts", new JTSCapability())
                .addRequirements("org.wildfly.transactions", "org.wildfly.iiop.orb",
"org.wildfly.iiop.corba-naming")
                .build();
```

When your capability is registered with the system, the WildFly Core kernel will automatically register any static hard requirements declared this way.

Registering a requirement for a dynamically named capability

If the capability you require is dynamically named, usually your capability's resource will include an attribute whose value is the dynamic part of the required capability's name. You should declare this fact in the `AttributeDefinition` for the attribute using the `SimpleAttributeDefinitionBuilder.setCapabilityReference` method.

For example, the WildFly "remoting" subsystem's "org.wildfly.remoting.connector" capability has a requirement for a dynamically named socket-binding capability:

```
public class ConnectorResource extends SimpleResourceDefinition {

    . . .

    static final String SOCKET_CAPABILITY_NAME = "org.wildfly.network.socket-binding";
    static final RuntimeCapability<Void> CONNECTOR_CAPABILITY =
            RuntimeCapability.Builder.of("org.wildfly.remoting.connector", true)
                    .build();

    . . .

    static final SimpleAttributeDefinition SOCKET_BINDING =
            new SimpleAttributeDefinitionBuilder(CommonAttributes.SOCKET_BINDING,
ModelType.STRING, false)

.addAccessConstraint(SensitiveTargetAccessConstraintDefinition.SOCKET_BINDING_REF)
                .setCapabilityReference(SOCKET_CAPABILITY_NAME, CONNECTOR_CAPABILITY)
                .build();
```

If the "add" operation handler for your resource extends `AbstractAddStepHandler` and the handler for `write-attribute` extends `AbstractWriteAttributeHandler`, the declaration above is sufficient to ensure that the appropriate capability requirement will be registered when the attribute is modified.

Depending upon a service provided by another capability

Once the requirement for the capability is registered, your `OperationStepHandler` can use the `OperationContext` to discover the name of the service provided by the required capability.

For example, the "add" handler for a remoting connector uses the `OperationContext` to find the name of the needed {{SocketBinding} service:

```
final String socketName = ConnectorResource.SOCKET_BINDING.resolveModelAttribute(context,
fullModel).asString();
        final ServiceName socketBindingName =
context.getCapabilityServiceName(ConnectorResource.SOCKET_CAPABILITY_NAME, socketName,
SocketBinding.class);
```

That service name is then used to add a dependency on the `SocketBinding` service to the remoting connector service.

If the required capability isn't dynamically named, `OperationContext` exposes an overloaded `getCapabilityServiceName` variant. For example, if a capability requires a remoting Endpoint:

```
ServiceName endpointService = context.getCapabilityServiceName("org.wildfly.remoting.endpoint",
Endpoint.class);
```

Using a custom integration API provided by another capability

In your `Stage.RUNTIME` handler, use `OperationContext.getCapabilityRuntimeAPI` to get a reference to the required capability's custom integration API. Then use it as necessary.

```
List<String> orbInitializers = new ArrayList<String>();
        . . .
        JTSCapability jtsCapability =
context.getCapabilityRuntimeAPI(IIOPExtension.JTS_CAPABILITY, JTSCapability.class);
        orbInitializers.addAll(jtsCapability.getORBInitializerClasses());
```

Runtime-only requirements

If your capability has a runtime-only requirement for another capability, that means that if that capability is present in `Stage.RUNTIME` you'll use it, and if not you won't. There is nothing about the configuration of your capability that triggers the need for the other capability; you'll just use it if it's there.

In this case, use `OperationContext.hasOptionalCapability` in your `Stage.RUNTIME` handler to check if the capability is present:

```
protected void performRuntime(final OperationContext context, final ModelNode operation, final
ModelNode model) throws OperationFailedException {

        ServiceName myServiceName = MyResource.FOO_CAPABILITY.getCapabilityServiceName();
        Service<DataSource> myService = createService(context, model);
        ServiceBuilder<DataSource> builder = context.getTarget().addService(myServiceName,
myService);

        // Inject a "Bar" into our "Foo" if bar capability is present
        if (context.hasOptionalCapability("com.example.bar",
MyResource.FOO_CAPABILITY.getName(), null) {
            ServiceName barServiceName = context.getCapabilityServiceName("com.example.bar",
Bar.class);
            builder.addDependency(barServiceName, Bar.class, myService.getBarInjector());
        }

        builder.install();
    }
```

The WildFly Core kernel will not register a requirement for the "com.example.bar" capability, so if a configuration change occurs that means that capability will no longer be present, that change will not be rolled back. Because of this, runtime-only requirements can only be used with capabilities that declare in their contract that they support such use.

Using a capability in a DeploymentUnitProcessor

A `DeploymentUnitProcessor` is likely to have a need to interact with capabilities, in order to create service dependencies from a deployment service to a capability provided service or to access some aspect of a capability's custom integration API that relates to deployments.

If a `DeploymentUnitProcessor` associated with a capability implementation needs to utilize its own capability object, the `DeploymentUnitProcessor` authors should simply provide it with a reference to the `RuntimeCapability` instance. Service name lookups or access to the capabilities custom integration API can then be performed by invoking the methods on the `RuntimeCapability`.

If you need to access service names or a custom integration API associated with a different capability, you will need to use the `org.jboss.as.controller.capability.CapabilityServiceSupport` object associated with the deployment unit. This can be found as an attachment to the `DeploymentPhaseContext`:

```
class MyDUP implements DeploymentUntiProcessor {

    public void deploy(DeploymentPhaseContext phaseContext) throws
DeploymentUnitProcessingException {

        AttachmentKey<CapabilityServiceSupport> key =
org.jboss.as.server.deployment.Attachments.DEPLOYMENT_COMPLETE_SERVICES;
        CapabilityServiceSupport capSvcSupport = phaseContext.getAttachment(key);
```

Once you have the `CapabilityServiceSupport` you can use it to look up service names:

```
ServiceName barSvcName = capSvcSupport.getCapabilityServiceName("com.example.bar");
        // Determine what 'baz' the user specified in the deployment descriptor
        String bazDynamicName = getSelectedBaz(phaseContext);
        ServiceName bazSvcName = capSvcSupport.getCapabilityServiceName("com.example.baz",
bazDynamicName);
```

It's important to note that when you request a service name associated with a capability, the `CapabilityServiceSupport` will give you one regardless of whether the capability is actually registered with the kernel. If the capability isn't present, any service dependency your DUP creates using that service name will eventually result in a service start failure, due to the missing dependency. This behavior of not failing immediately when the capability service name is requested is deliberate. It allows deployment operations that use the `rollback-on-runtime-failure=false` header to successfully install (but not start) all of the services related to a deployment. If a subsequent operation adds the missing capability, the missing service dependency problem will then be resolved and the MSC service container will automatically start the deployment services.

You can also use the `CapabilityServiceSupport` to obtain a reference to the capability's custom integration API:

```
// We need custom integration with the baz capability beyond service injection
        BazIntegrator bazIntegrator;
        try {
            bazIntegrator = capSvcSupport.getCapabilityRuntimeAPI("com.example.baz",
bazDynamicName, BazIntegrator.class);
        } catch (NoSuchCapabilityException e) {
            //
            String msg = String.format("Deployment %s requires use of the 'bar' capability but
it is not currently registered",
                                       phaseContext.getDeploymentUnit().getName());
            throw new DeploymentUnitProcessingException(msg);
        }
```

Note that here, unlike the case with service name lookups, the `CapabilityServiceSupport` will throw a checked exception if the desired capability is not installed. This is because the kernel has no way to satisfy the request for a custom integration API if the capability is not installed. The `DeploymentUnitProcessor` will need to catch and handle the exception.

Detailed API

The WildFly Core kernel's API for using capabilities is covered in detail in the javadoc for the RuntimeCapability and RuntimeCapability.Builder classes and the OperationContext and CapabilityServiceSupport interfaces.

Many of the methods in `OperationContext` related to capabilities have to do with registering capabilities or registering requirements for capabilities. Typically non-kernel developers won't need to worry about these, as the abstract `OperationStepHandler` implementations provided by the kernel take care of this for you, as described in the preceding sections. If you do find yourself in a situation where you need to use these in an extension, please read the javadoc thoroughly.

12 Common

12.1 All WildFly documentation

There are several guides in the WildFly documentation series. This list gives an overview of each of the guides:

*Getting Started Guide - Explains how to download and start WildFly.
*Getting Started Developing Applications Guide - Talks you through developing your first applications on WildFly, and introduces you to JBoss Tools and how to deploy your applications.
*JavaEE 6 Tutorial - A Java EE 6 Tutorial.
*Admin Guide - Tells you how to configure and manage your WildFly instances.
*Developer Guide - Contains concepts that you need to be aware of when developing applications for WildFly. Classloading is explained in depth.
*High Availability Guide - Reference guide for how to set up clustered WildFly instances.
*Extending WildFly - A guide to adding new functionality to WildFly.

13 Testsuite

13.1 JBoss AS 7 Testsuite

Where to go next?

- WildFly Integration Testsuite User Guide if you changed JBoss AS 7 code and want to check for regressions.
- AS 7 Testsuite Harness Developer Guide to learn how the testsuite works (shell scripts, Ant scripts, pom.xml files)
- AS 7 Testsuite Test Developer Guide if you want to add a new test case to the testsuite (to increase code coverage or to reproduce a bug)

13.2 WildFly Testsuite Overview

This document will detail the implementation of the testsuite Integration submodule as it guides you on adding your own test cases.

The WildFly integration test suite has been designed with the following goals:

- support execution of all identified test case use cases
- employ a design/organization which is scalable and maintainable
- provide support for the automated measurement of test suite quality (generation of feature coverage reports, code coverage reports)

In addition, these requirements were considered:

- identifying distinct test case runs of the same test case with a different set of client side parameters and server side parameters
- separately maintaining server side execution results (e.g. logs, the original server configuration) for post-execution debugging
- running the testsuite in conjunction with a debugger
- the execution of a single test (for debugging purposes)
- running test cases against different container modes (managed in the main, but also remote and embedded)
- configuring client and server JVMs separately (e.g., IPv6 testing)

13.2.1 Test Suite Organization

The testsuite module has a small number of submodules:

- **benchmark** - holds all benchmark tests intended to assess relative performance of specific feature
- **domain** - holds all domain management tests
- **integration** - holds all integration tests
- **stress** - holds all stress tests

It is expected that test contributions fit into one of these categories.

The pom.xml file located in the testsuite module is inherited by all submodules and is used to do the following:

- set defaults for common testsuite system properties (which can then be overridden on the command line)
- define dependencies common to all tests (Arquillian, junit or testng, and container type)
- provide a workaround for @Resource(lookup=...) which requires libraries in jbossas/endorsed

It should not:

- define module-specific server configuration build steps
- define module-specific surefire executions

These elements should be defined in logical profiles associated with each logical grouping of tests; e.g., in the pom for the module which contains the tests. The submodule poms contain additional details of their function and purpose as well as expanded information as shown in this document.

13.2.2 Profiles

You should not activate the abovementioned profiles by -P, because that disables other profiles which are activated by default.

Instead, you should always use activating properties, which are in parenthesis in the lists below.

Testsuite profiles are used to group tests into logical groups.

- all-modules.module.profile (all-modules)
- integration.module.profile (integration.module)
- compat.module.profile (compat.module)
- domain.module.profile (domain.module)
- benchmark.module.profile (benchmark.module)
- stress.module.profile (stress.module)

They also pprepare WildFly instances and resources for respective testsuite submodules.

- jpda.profile - sets surefire.jpda.args (debug)
- ds.profile -sets database properties and prepares the datasource (ds=<db id>)
    - Has related database-specific profiles, like mysql51.profile etc.

Integration testsuite profiles configure surefire executions.

- smoke.integration.tests.profile
- basic.integration.tests.profile
- clustering.integration.tests.profile

13.2.3 Integration tests

Smoke -Dts.smoke, -Dts.noSmoke

Contains smoke tests.

Runs by default; use -Dts.noSmoke to prevent running.

Tests should execute quickly.

Divided into two Surefire executions:

- One with full profile
- Second with web profile (majority of tests).

Basic -Dts.basic

Basic integration tests - those which do not need special configuration like cluster.

Divided into three Surefire executions:

- One with full profile,
- Second with web profile (majority of tests).
- Third with web profile, but needs to be run after server restart to check whether persistent data are really persisted.

Cluster -Dts.clust

Sets up a cluster of two nodes.

IIOP -Dts.iiop

XTS -Dts.XTS

Multinode -Dts.multinode

13.3 WildFly Integration Testsuite User Guide

See also: WildFly Testsuite Test Developer Guide

Target Audience: Those interested in running the testsuite or a subset thereof, with various configuration options.

13.3.1 Running the testsuite

The tests can be run using:

- `build.sh` or `build.bat`, as a part of WildFly build.
  - By default, only smoke tests are run. To run all tests, run build.sh install -DallTests.
- `integration-tests.sh` or `integration-tests.bat`, a convenience script which uses bundled Maven (currently 3.0.3), and runs all parent testsuite modules (which configure the AS server).
- pure maven run, using `mvn install`.

The scripts are wrappers around Maven-based build. Their arguments are passed to Maven (with few exceptions described below). This means you can use:

- build.sh (defaults to install)
- build.sh install
- build.sh clean install
- integration-tests.sh install
- ...etc.

Supported Maven phases

Testsuite actions are bounds to various Maven phases up to `verify`. Running the build with earlier phases may fail in the submodules due to missed configuration steps. Therefore, the only Maven phases you may safely run, are:

- clean
- install
- site

The `test` phase is not recommended to be used for scripted jobs as we are planning to switch to the `failsafe` plugin bound to the `integration-test` and `verify` phases. See WFLY-625 and WFLY-228.

Testsuite structure

testsuite
- integration
  - smoke
  - basic
  - clust
  - iiop
  - multinode
  - xts
- compat
- domain
- mixed-domain
- stress
- benchmark

Test groups

To define groups of tests to be run, these properties are available:

- `-DallTests` - Runs all subgroups.
- `-DallInteg` - Runs all integration tests. Same as `cd testsuite/integration; mvn clean install -DallTests`
- `-Dts.integ` - Basic integration + clustering tests.
- `-Dts.clust` - Clustering tests.
- `-Dts.iiop` - IIOP tests.
- `-Dts.multinode` - Tests with many nodes.
- `-Dts.manualmode` - Tests with manual mode Arquillian containers.
- `-Dts.bench` - Benchmark tests.
- `-Dts.stress` - Stress tests.
- `-Dts.domain` - Domain mode tests.
- `-Dts.compat` - Compatibility tests.

13.3.2 Examples

- `integration-tests.sh [install]  -- Runs smoke tests.`
- `integration-tests.sh clean install  -- Cleans the target directory, then runs smoke tests.`
- `integration-tests.sh install -Dts.smoke  -- Same as above.`
- `integration-tests.sh install -DallTests  -- Runs all testsuite tests.`
- `integration-tests.sh install -Dts.stress  -- Runs smoke tests and stress tests.`
- `integration-tests.sh install -Dts.stress -Dts.noSmoke  -- Runs stress tests only.`

Pure maven - if you prefer not to use scripts, you may achieve the same result with:

- `mvn ... -rf testsuite`

The `-rf ...` parameter stands for "resume from" and causes Maven to run the specified module *and all successive*.

It's possible to run only a single module (provided the ancestor modules were already run to create the AS copies) :

- `mvn ... -pl testsuite/integration/cluster`

The `-pl ...` parameter stands for "project list" and causes Maven to run the specified module *only*.

Output to console

```
-DtestLogToFile
```

Other options

`-DnoWebProfile` - Run all tests with the *full* profile (`standalone-full.xml`). By default, most tests are run under *web* profile (`standalone.xml`).

`-Dts.skipTests` - Skip testsuite's tests. Defaults to the value of `-DskipTests`, which defaults to `false`. To build AS, skip unit tests and run testsuite, use `-DskipTests -Dts.skipTests=false`.

Timeouts

Surefire execution timeout

Unfortunatelly, no math can be done in Maven, so instead of applying a timeout ratio, you need to specify timeout manually for Surefire.

```
-Dsurefire.forked.process.timeout=900
```

test timeout ratios

Ratio in prercent - 100 = default, 200 = two times longer timeouts for given category.

Currently we have five different ratios. Later, it could be replaced with just one generic, one for database and one for deployment operations.

```
-Dtimeout.ratio.fsio=100
-Dtimeout.ratio.netio=100
-Dtimeout.ratio.memio=100
-Dtimeout.ratio.proc=100
-Dtimeout.ratio.db=100
```

Running a single test (or specified tests)

Single test is run using **-Dtest=...** . Examples:

- `./integration-tests.sh install -Dtest='*Clustered*' -Dintegration.module -Dts.clust`
- `./integration-tests.sh clean install -Dtest=org/jboss/as/test/integration/ejb/async/*TestCase.java -Dintegration.module -Dts.basic`
- `cd testsuite; mvn install -Dtest='*Clustered*' -Dts.basic  # No need for -Dintegration.module - integration module is active by default.`

The same shortcuts listed in "Test groups" may be used to activate the module and group profile.

Note that `-Dtest=` overrides `<includes>` and `<exludes>` defined in pom.xml, so do not rely on them when using wildcards - all compiled test classes matching the wildcard will be run.

Which Surefire execution is used?

Due to Surefire's design flaw, tests run multiple times if there are multiple surefire executions.
To prevent this, if `-Dtest=`... is specified, non-default executions are disabled, and standalone-full is used for all tests.
If you need it other way, you can overcome that need:

- `basic-integration-web.surefire with standalone.xml - Configure standalone.xml to be used as server config.`
- `basic-integration-non-web.surefire - For tests included here, technically nothing changes.`
- `basic-integration-2nd.surefire - Simply run the second test in another invocation of Maven.`

Running against existing AS copy (not the one from build/target/jboss-as-*)

-Djboss.dist=<path/to/jboss-as> will tell the testsuite to copy that AS into submodules to run the tests against.

For example, you might want to run the testsuite against AS located in `/opt/wildfly-8` :

```
./integration-tests.sh -DallTests -Djboss.dist=/opt/wildfly-8
```

The difference between jboss.dist and jboss.home:

jboss.dist is the location of the tested binaries. It gets copied to testsuite submodules.

jboss.home is internally used and points to those copied AS instances (for multinode tests, may be even different for each AS started by Arquillian).

Running against a running JBoss AS instance

Arquillian's WildFly 8 container adapter allows specifying `allowConnectingToRunningServer` in `arquillian.xml`, which makes it check whether AS is listening at `managementAddress:managementPort`, and if so, it uses that server instead of launching a new one, and doesn't shut it down at the end.

All arquillian.xml's in the testsuite specify this parameter. Thus, if you have a server already running, it will be re-used.

Running against JBoss Enterprise Application Platform (EAP) 6.0

To run the testsuite against AS included JBoss Enterprise Application Platform 6.x (EAP), special steps are needed.

Assuming you already have the sources available, and the distributed EAP maven repository unzipped in e.g. `/opt/jboss/eap6-maven-repo/` :

1) Configure maven in settings.xml to use only the EAP repository. This repo contains all artifacts necessary for building EAP, including maven plugins.
The build (unlike running testsuite) may be done offline.
The recommended way of configuring is to use special settings.xml, not your local one (typically in .m2/settings.xml).

```
   <mirror>
      <id>eap6-mirror-setting</id>
      <mirrorOf>
         *,!central-eap6,!central-eap6-plugins,!jboss-public-eap6,!jboss-public-eap6-plugins
      </mirrorOf>
      <name>Mirror Settings for EAP 6 build</name>
      <url>file:///opt/jboss/eap6-maven-repo</url>
    </mirror>
  </mirrors>
```

2) Build EAP. You won't use the resulting EAP build, though. The purpose is to get the artifacts which the testsuite depends on.

```
mvn clean install -s settings.xml -Dmaven.repo.local=local-repo-eap
```

3) Run the testsuite. Assuming that EAP is located in `/opt/eap6`, you would run:

```
./integration-tests.sh -DallTests -Djboss.dist=/opt/eap6
```

For further information on building EAP and running the testsuite against it, see the official EAP documentation (link to be added).

How-to for EAP QA can be found here (Red Hat internal only).

Running with a debugger

Argument	What will start with debugger	Default port	Port change arg.
-Ddebug	AS instances run by Arquillian	8787	-Das.debug.port=...
-Djpda	alias for -Ddebug		
-DdebugClient	Test JVMs (currently Surefire)	5050	-Ddebug.port.surefire=...
-DdebugCLI	AS CLI	5051	-Ddebug.port.cli=...

Examples

```
./integration-tests.sh install -DdebugClient -Ddebug.port.surefire=4040

...

-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Listening for transport dt_socket at address: 4040
```

```
./integration-tests.sh install -DdebugClient -Ddebug.port.surefire

...

-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Listening for transport dt_socket at address: 5050
```

```
./integration-tests.sh install -Ddebug
```

```
./integration-tests.sh install -Ddebug -Das.debug.port=5005
```

JBoss AS is started by Arquillian, when the first test which requires given instance is run. Unless you pass **-DtestLogToFile=false, there's (currently) no challenge text in the console**; it will look like the first test is stuck. This is being solved in http://jira.codehaus.org/browse/SUREFIRE-781.

Depending on which test group(s) you run, multiple AS instances may be started. In that case, you need to attach the debugger multiple times.

Running tests with custom database

To run with different database, specify the `-Dds` and use these properties (with the following defaults):

```
-Dds.jdbc.driver=
-Dds.jdbc.driver.version=
-Dds.jdbc.url=
-Dds.jdbc.user=test
-Dds.jdbc.pass=test
-Dds.jdbc.driver.jar=${ds.db}-jdbc-driver.jar
```

`driver` is JDBC driver class. JDBC `url`, `user` and `pass` is as expected.

`driver.version` is used for automated JDBC driver downloading. Users can set up internal Maven repository hosting JDBC drivers, with artifacts with

GAV = `jdbcdrivers:${ds.db}:${ds.jdbc.driver.version}`

Internally, JBoss has such repo at http://nexus.qa.jboss.com:8081/nexus/content/repositories/thirdparty/jdbcdrivers/ .

The `ds.db` value is set depending on ds. E.g. `-Dds=mssql2005` sets `ds.db=mssql` (since they have the same driver). `-Dds.db` may be overriden to use different driver.

~~In case you don't want to use such driver, set just -Dds.db= (empty) and provide the driver to the AS manually.~~
Not supported; work in progress on parameter to provide JDBC Driver jar.

Default values

For WildFly continuous integration, there are some predefined values for some of databases, which can be set using:

```
-Dds.db=<database-identifier>
```

Where database-identifier is one of: `h2`, `mysql51`

Running tests with IPv6

`-Dipv6` - Runs AS with `-Djava.net.preferIPv4Stack=false -Djava.net.preferIPv6Addresses=true`

and the following defaults, overridable by respective parameter:

Parameter	IPv4 default	IPv6 default	
-Dnode0	127.0.0.1	::1	Single-node tests.
-Dnode1	127.0.0.1	::1	Two-node tests (e.g. cluster) use this for the 2nd node.
-Dmcast	230.0.0.4	ff01::1	ff01::1 is IPv6 Node-Local scope mcast addr.
-Dmcast.jgroupsDiag	224.0.75.75	ff01::2	JGroups diagnostics multicast address.
-Dmcast.modcluster	224.0.1.105	ff01::3	mod_cluster multicast address.

Values are set in AS configuration XML, replaced in resources (like ejb-jar.xml) and used in tests.

Running tests with security manager / custom security policy

`-Dsecurity.manager` - Run with default policy.

`-Dsecurity.policy=<path>` - Run with the given policy.

`-Dsecurity.manager.other=<set of Java properties>` - Run with the given properties. Whole set is included in all server startup parameters.

Example:

```
./integration-tests.sh clean install -Dintegration.module -DallTests \
\"-Dsecurity.manager.other=-Djava.security.manager \
-Djava.security.policy==$(pwd)/testsuite/shared/src/main/resources/secman/permitt_all.policy \
-Djava.security.debug=access:failure \"
```

Notice the \" quotes delimiting the whole `-Dsecurity.manager.other` property.

Creating test reports

Test reports are created in the form known from EAP 5. To create them, simply run the testsuite, which will create Surefire XML files.

Creation of the reports is bound to the `site` Maven phase, so it must be run separatedly afterwards. Use one of these:

```
./integration-tests.sh site
```

```
cd testsuite; mvn site
```

```
mvn -pl testsuite site
```

Note that it will take all test results under `testsuite/integration/` - the pattern is `**/*TestCase.xml`, without need to specify `-DallTests`.

Creating coverage reports

Jira: https://issues.jboss.org/browse/WFLY-585

Coverage reports are created by JaCoCo.

During the integration tests, Arquillian is passed a JVM argument which makes it run with JaCoCo agent, which records the executions into `${basedir}/target/jacoco` .

In the `site` phase, a HTML, XML and CSV reports are generated. That is done using `jacoco:report` Ant task in `maven-ant-plugin` since JaCoCo's maven report goal doesn't support getting classes outside target/classes.

Usage

```
./build.sh clean install -DskipTests
./integration-tests.sh clean install -DallTests -Dcoverage
./integration-tests.sh site -DallTests -Dcoverage ## Must run in separatedly.
```

Alternative:

```
mvn clean install -DskipTests
mvn -rf testsuite clean install -DallTests -Dcoverage
mvn -rf testsuite site -DallTests -Dcoverage
```

Cleaning the project

To have most stable build process, it should start with:

- clean target directories
- only central Maven repo configured
- clean local repository or at least:
  - free of artefacts to be built
  - free of dependencies to be used (especially snapshots)

To use , you may use these commands:

```
mvn clean install -DskipTests -DallTests  ## ...to clean all testsuite modules.
mvn dependency:purge-local-repository build-helper:remove-project-artifact
-Dbuildhelper.removeAll
```

In case the build happens in a shared environment (e.g. network disk), it's recommended to use local repository:

```
cp /home/hudson/.m2/settings.xml .
sed
"s|<settings>|<settings><localRepository>/home/ozizka/hudson-repos/$JOBNAME</localRepository>|"
-i settings.xml
```

Or:

```
mvn clean install ... -Dmaven.repo.local=localrepo
```

See also https://issues.jboss.org/browse/WFLY-628.

13.3.3 Troubleshooting Common Issues

Timeouts

May happen especially on slower computers. Try setting a different timeout (in seconds) :

-Dsurefire.forked.process.timeout=9999

"Server already running"

Known issue: JBPAPP-8368 Arquillian should wait until a port is free after AS JVM process ends to prevent "port in use".

~~Currently, the only solution is to re-run.~~ This has been fixed in 7.1.2, see pull request https://github.com/jbossas/jboss-as/pull/1999 .

Database failures

Build gets stuck at first test of a module

If you use NFS (Network file system), it might be a file locking issue.

Try running using a local disk.

13.4 WildFly Testsuite Harness Developer Guide

Audience: Whoever wants to change the testsuite harness

JIRA: WFLY-576

13.4.1 Testsuite requirements

http://community.jboss.org/wiki/ASTestsuiteRequirements will probably be merged here later.

13.4.2 Adding a new maven plugin

The plugin version needs to be specified in jboss-parent. See https://github.com/jboss/jboss-parent-pom/blob/master/pom.xml#L65 .

13.4.3 Shortened Maven run overview

See Shortened Maven Run Overview.

13.4.4 How the AS instance is built

See How the JBoss AS instance is built and configured for testsuite modules.

13.4.5 Properties and their propagation

Propagated to tests through arquillian.xml:

```
<property name="javaVmArguments">${server.jvm.args}</property>
```

TBD: https://issues.jboss.org/browse/ARQ-647

JBoss AS instance dir

integration/pom.xml

(currently nothing)

***-arquillian.xml**

```
<container qualifier="jboss" default="true">
    <configuration>
        <property name="jbossHome">${basedir}/target/jbossas</property>
```

Server JVM arguments

```
<surefire.memory.args>-Xmx512m -XX:MaxPermSize=256m</surefire.memory.args>
    <surefire.jpda.args></surefire.jpda.args>
    <surefire.system.args>${surefire.memory.args} ${surefire.jpda.args}</surefire.system.args>
```

IP settings

- `${ip.server.stack}` - used in `<systemPropertyVariables> / <server.jvm.args>` which is used in `*-arquillian.xml`.

Testsuite directories

- `${jbossas.ts.integ.dir}`
- `${jbossas.ts.dir}`
- `${jbossas.project.dir}`

Clustering properties

- node0
- node1

13.4.6 Debug parameters propagation

```
<surefire.jpda.args></surefire.jpda.args>       - default

<surefire.jpda.args>-Xrunjdwp:transport=dt_socket,address=${as.debug.port},server=y,suspend=y</sur
- activated by -Ddebug or -Djpda


testsuite/pom.xml:        <surefire.system.args>... ${surefire.jpda.args}
...</surefire.system.args>
testsuite/pom.xml:                        <jboss.options>${surefire.system.args}</jboss.options>

testsuite/integration/pom.xml:     <server.jvm.args>${surefire.system.args}
${jvm.args.ip.server} ${jvm.args.security} ${jvm.args.timeouts} -Dnode0=${node0} -Dnode1=


integration/pom.xml:
<server.jvm.args>${surefire.system.args} ${jvm.args.ip.server} ${jvm.args.security}
${jvm.args.timeouts} -Dnode0=${node0} -Dnode1=${node1} -DudpGroup=${udpGroup}
${jvm.args.dirs}</server.jvm.args>

arquillian.xml:
<property name="javaVmArguments">${server.jvm.args}
-Djboss.inst=${basedir}/target/jbossas</property>
```

13.4.7 How the JBoss AS instance is built and configured for testsuite modules.

Refer to Shortened Maven Run Overview to see the mentioned build steps.

1) AS instance is copied from `${jboss.dist}` to `testsuite/target/jbossas`.
   Defaults to AS which is built by the project (`build/target/jboss-as-*`).

2)

testsuite/pom.xml:

from ${jboss.home} to ${basedir}/target/jbossas
phase generate-test-resources: resource-plugin, goal copy-resources

testsuite/integration/pom.xml:

phase process-test-resources: antrun-plugin:

```
<ant antfile="$\{basedir}/src/test/scripts/basic-integration-build.xml">
    <target name="build-basic-integration"/>
    <target name="build-basic-integration-jts"/>
</ant>
```

Which invokes

```
<target name="build-basic-integration" description="Builds server configuration for
basic-integration tests">
      <build-server-config name="jbossas"/>
```

Which invokes

```
<!-- Copy the base distribution. -->
<!-- We exclude modules and bundles as they are read-only and we locate the via sys props. -->
<copy todir="@{output.dir}/@{name}">
    <fileset dir="@{jboss.dist}">
        <exclude name="**/modules/**"/>
        <exclude name="**/bundles/**"/>
    </fileset>
</copy>

<!-- overwrite with configs from test-configs and apply property filtering -->
<copy todir="@{output.dir}/@{name}" overwrite="true" failonerror="false">
    <fileset dir="@{test.configs.dir}/@{name}"/>
    <filterset begintoken="${" endtoken="}">
        <filter token="node0" value="${node0}"/>
        <filter token="node1" value="${node1}"/>
        <filter token="udpGroup" value="${udpGroup}"/>
        <filter-elements/>
    </filterset>
</copy>
```

Arquillian config file location

```
-Darquillian.xml=some-file-or-classpath-resource.xml
```

13.4.8 Plugin executions matrix

x - runs in this module
xx - runs in this and all successive modules
x! - runs but should not.

initialize									
	TS	integ	smoke	basic	clust	iiop	comp	domain	be

maven-help-plugin	xx	x	x	x	x	x	x	x	x
properties-maven-plugin:write-project-properties	x								
maven-antrun-plugin:1.6:run (banner)									
process-resources									
maven-resources-plugin:2.5:resources (default-resources)	xx								
maven-dependency-plugin:2.3:copy (copy-annotations-endorsed)	xx!								
compile									
maven-compiler-plugin:2.3.2:compile (default-compile)	xx								
generate-test-resources									
maven-resources-plugin:2.5:copy-resources (build-jbossas.server)	xx!								
Should be:	x								
maven-resources-plugin:2.5:copy-resources (ts.copy-jbossas)		x							
maven-resources-plugin:2.5:copy-resources (ts.copy-jbossas.groups)		x	x	x	?	?	?	!	
Should be:		xx	x	x	x	x	x	x	x
process-test-resources									
maven-resources-plugin:2.5:testResources (default-testResources)	xx								
maven-antrun-plugin:1.6:run (build-smoke.server)			x						
maven-antrun-plugin:1.6:run (prepare-jars-basic-integration.server)				x					
maven-antrun-plugin:1.6:run (build-clustering.server)					x	x!?			
test-compile									
maven-compiler-plugin:2.3.2:testCompile (default-testCompile)	xx								
xml-maven-plugin:1.0:transform (update-ip-addresses-jbossas.server)	x								

maven-antrun-plugin:1.6:run (build-jars)							x		
test									
maven-surefire-plugin:2.10:test (smoke-full.surefire)									
maven-surefire-plugin:2.10:test (smoke-web.surefire)									
maven-surefire-plugin:2.10:test (default-test)					x	x	x	x	
maven-surefire-plugin:2.10:test (basic-integration-default-full.surefire)				x					
maven-surefire-plugin:2.10:test (basic-integration-default-web.surefire)				x					
maven-surefire-plugin:2.10:test (basic-integration-2nd.surefire)				x					
maven-surefire-plugin:2.10:test (tests-clust-multi-node-unm...surefire)					x				
maven-surefire-plugin:2.10:test (tests-clustering-single-node.surefire)					x				
maven-surefire-plugin:2.10:test (tests-clustering-multi-node.surefire)					x				
maven-surefire-plugin:2.10:test (tests-iiop-multi-node.surefire)						x			
package									
maven-jar-plugin:2.3.1:jar (default-jar)	xx!								
maven-source-plugin:2.1.2:jar-no-fork (attach-sources)	x								
install									
maven-install-plugin:2.3.1:install (default-install)	xx!								
	TS	integ	smoke	basic	clust	iiop	comp	domain	be

13.4.9 Shortened Maven Run Overview

How to get it

```
./integration-tests.sh clean install -DallTests | tee TS.txt | testsuite/tools/runSummary.sh
```

How it's done

Run this script on the output of the AS7 testsuite run:

```
##  Cat the file or stdin if no args,
##  filter only interesting lines - plugin executions and modules separators,
##  plus Test runs summaries,
##  and remove the boring plugins like enforcer etc.

cat $1 \
 | egrep ' --- |Building| ---------|Tests run: | T E S T S' \
 | grep -v 'Time elapsed'
 | sed 's|Tests run:|                Tests run:|' \
 | grep -v maven-clean-plugin \
 | grep -v maven-enforcer-plugin \
 | grep -v buildnumber-maven-plugin \
 | grep -v maven-help-plugin \
 | grep -v properties-maven-plugin:.*:write-project-properties \
;
```

You'll get an overview of the run.

Example output with comments.

```
ondra@ondra-redhat: ~/work/AS7/ozizka-as7 $  ./integration-tests.sh clean install -DallTests |
tee TS.txt | testsuite/tools/runSummary.sh
[INFO] ------------------------------------------------------------------------
[INFO] ------------------------------------------------------------------------
[INFO] Building JBoss Application Server Test Suite: Aggregator 7.1.0.CR1-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO] --- maven-dependency-plugin:2.3:copy (copy-annotations-endorsed) @ jboss-as-testsuite ---
              Copies org.jboss.spec.javax.annotation:jboss-annotations-api_1.1_spec to
${project.build.directory}/endorsed .
              Inherited - needed for compilation of all submodules.

[INFO] --- maven-resources-plugin:2.5:copy-resources (build-jbossas.server) @ jboss-as-testsuite
---
              Copies ${jboss.home} to target/jbossas .  TODO: Should be jboss.dist.

[INFO] --- xml-maven-plugin:1.0:transform (update-ip-addresses-jbossas.server) @
jboss-as-testsuite ---
              Changes IP addresses used in server config files -
```

```
              applies ${xslt.scripts.dir}/changeIPAddresses.xsl on
${basedir}/target/jbossas/standalone/configuration/standalone-*.xml
              Currently inherited, IMO should not be.

[INFO] --- maven-source-plugin:2.1.2:jar-no-fork (attach-sources) @ jboss-as-testsuite ---
              TODO: Remove

[INFO] --- maven-install-plugin:2.3.1:install (default-install) @ jboss-as-testsuite ---

[INFO] ------------------------------------------------------------------------
[INFO] Building JBoss Application Server Test Suite: Integration Aggregator 7.1.0.CR1-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO] --- maven-dependency-plugin:2.3:copy (copy-annotations-endorsed) @
jboss-as-testsuite-integration-agg ---
[INFO] --- maven-resources-plugin:2.5:copy-resources (build-jbossas.server) @
jboss-as-testsuite-integration-agg ---
              TODO: Remove
[INFO] --- maven-resources-plugin:2.5:copy-resources (ts.copy-jbossas) @
jboss-as-testsuite-integration-agg ---
[INFO] --- maven-resources-plugin:2.5:copy-resources (ts.copy-jbossas.groups) @
jboss-as-testsuite-integration-agg ---
[INFO] --- xml-maven-plugin:1.0:transform (update-ip-addresses-jbossas.server) @
jboss-as-testsuite-integration-agg ---
              TODO: Remove
[INFO] --- maven-source-plugin:2.1.2:jar-no-fork (attach-sources) @
jboss-as-testsuite-integration-agg ---
              TODO: Remove
[INFO] --- maven-install-plugin:2.3.1:install (default-install) @
jboss-as-testsuite-integration-agg ---
[INFO] ------------------------------------------------------------------------
[INFO] Building JBoss AS Test Suite: Integration - Smoke 7.1.0.CR1-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO] --- maven-dependency-plugin:2.3:copy (copy-annotations-endorsed) @
jboss-as-testsuite-integration-smoke ---
[INFO] --- maven-resources-plugin:2.5:resources (default-resources) @
jboss-as-testsuite-integration-smoke ---
              TODO: Remove
[INFO] --- maven-compiler-plugin:2.3.2:compile (default-compile) @
jboss-as-testsuite-integration-smoke ---
[INFO] --- maven-resources-plugin:2.5:copy-resources (build-jbossas.server) @
jboss-as-testsuite-integration-smoke ---
[INFO] --- maven-resources-plugin:2.5:copy-resources (ts.copy-jbossas.groups) @
jboss-as-testsuite-integration-smoke ---
[INFO] --- maven-resources-plugin:2.5:testResources (default-testResources) @
jboss-as-testsuite-integration-smoke ---
[INFO] --- xml-maven-plugin:1.0:transform (update-ip-addresses-jbossas.server) @
jboss-as-testsuite-integration-smoke ---
              TODO: Remove

[INFO] --- maven-antrun-plugin:1.6:run (build-smoke.server) @
jboss-as-testsuite-integration-smoke ---
     [echo] Building AS instance "smoke" from /home/ondra/work/EAP/EAP6-DR9 to
/home/ondra/work/AS7/ozizka-as7/testsuite/integration/smoke/target
              TODO: Should be running one level above!

[INFO] --- maven-compiler-plugin:2.3.2:testCompile (default-testCompile) @
jboss-as-testsuite-integration-smoke ---
[INFO] --- maven-surefire-plugin:2.10:test (smoke-full.surefire) @
```

```
jboss-as-testsuite-integration-smoke ---
 T E S T S
               Tests run: 4, Failures: 0, Errors: 4, Skipped: 0
```

Example output, unchanged

```
ondra@lenovo:~/work/AS7/ozizka-git$ ./integration-tests.sh clean install -DallTests | tee TS.txt
| testsuite/tools/runSummary.sh
SSCmeetingWestfordJan     [copy] Warning:
/home/ondra/work/AS7/ozizka-git/testsuite/integration/src/test/resources/test-configs/smoke does
not exist.
     [copy] Warning:
/home/ondra/work/AS7/ozizka-git/testsuite/integration/src/test/resources/test-configs/clustering-u
does not exist.
     [copy] Warning:
/home/ondra/work/AS7/ozizka-git/testsuite/integration/src/test/resources/test-configs/clustering-u
does not exist.
     [copy] Warning:
/home/ondra/work/AS7/ozizka-git/testsuite/integration/src/test/resources/test-configs/iiop-client
does not exist.
     [copy] Warning:
/home/ondra/work/AS7/ozizka-git/testsuite/integration/src/test/resources/test-configs/iiop-server
does not exist.
[INFO] ------------------------------------------------------------------------
[INFO] ------------------------------------------------------------------------
[INFO] Building JBoss Application Server Test Suite: Aggregator 7.1.0.Final-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO] --- maven-antrun-plugin:1.6:run (banner) @ jboss-as-testsuite ---
[INFO] --- maven-dependency-plugin:2.3:copy (copy-annotations-endorsed) @ jboss-as-testsuite ---
[INFO] --- maven-resources-plugin:2.5:copy-resources (build-jbossas.server) @ jboss-as-testsuite
---
[INFO] --- xml-maven-plugin:1.0:transform (update-ip-addresses-jbossas.server) @
jboss-as-testsuite ---
[INFO] --- maven-install-plugin:2.3.1:install (default-install) @ jboss-as-testsuite ---
[INFO] ------------------------------------------------------------------------
[INFO] Building JBoss Application Server Test Suite: Integration 7.1.0.Final-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO] --- maven-dependency-plugin:2.3:copy (copy-annotations-endorsed) @
jboss-as-testsuite-integration-agg ---
[INFO] --- maven-resources-plugin:2.5:copy-resources (build-jbossas.server) @
jboss-as-testsuite-integration-agg ---
[INFO] --- maven-resources-plugin:2.5:copy-resources (ts.copy-jbossas) @
jboss-as-testsuite-integration-agg ---
[INFO] --- maven-resources-plugin:2.5:copy-resources (ts.copy-jbossas.groups) @
jboss-as-testsuite-integration-agg ---
[INFO] --- maven-install-plugin:2.3.1:install (default-install) @
jboss-as-testsuite-integration-agg ---
[INFO] ------------------------------------------------------------------------
[INFO] Building JBoss Application Server Test Suite: Integration - Smoke 7.1.0.Final-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO] --- maven-dependency-plugin:2.3:copy (copy-annotations-endorsed) @
jboss-as-testsuite-integration-smoke ---
[INFO] --- maven-resources-plugin:2.5:resources (default-resources) @
jboss-as-testsuite-integration-smoke ---
```

```
[INFO] --- maven-compiler-plugin:2.3.2:compile (default-compile) @
jboss-as-testsuite-integration-smoke ---
[INFO] --- maven-resources-plugin:2.5:copy-resources (build-jbossas.server) @
jboss-as-testsuite-integration-smoke ---
[INFO] --- maven-resources-plugin:2.5:copy-resources (ts.copy-jbossas.groups) @
jboss-as-testsuite-integration-smoke ---
[INFO] --- maven-resources-plugin:2.5:testResources (default-testResources) @
jboss-as-testsuite-integration-smoke ---
[INFO] --- maven-antrun-plugin:1.6:run (build-smoke.server) @
jboss-as-testsuite-integration-smoke ---
     [echo] Building AS instance "smoke" from
/home/ondra/work/AS7/ozizka-git/testsuite/integration/smoke/../../../build/target/jboss-as-7.1.0.F
to /home/ondra/work/AS7/ozizka-git/testsuite/integration/smoke/target
[INFO] --- maven-compiler-plugin:2.3.2:testCompile (default-testCompile) @
jboss-as-testsuite-integration-smoke ---
[INFO] --- maven-surefire-plugin:2.10:test (smoke-full.surefire) @
jboss-as-testsuite-integration-smoke ---
 T E S T S
               Tests run: 4, Failures: 0, Errors: 0, Skipped: 0
[INFO] --- maven-surefire-plugin:2.10:test (smoke-web.surefire) @
jboss-as-testsuite-integration-smoke ---
 T E S T S
               Tests run: 116, Failures: 0, Errors: 0, Skipped: 6
[INFO] --- maven-jar-plugin:2.3.1:jar (default-jar) @ jboss-as-testsuite-integration-smoke ---
[INFO] Building jar:
/home/ondra/work/AS7/ozizka-git/testsuite/integration/smoke/target/jboss-as-testsuite-integration-
--- maven-install-plugin:2.3.1:install (default-install) @ jboss-as-testsuite-integration-smoke
---
[INFO] ------------------------------------------------------------------------
[INFO] Building JBoss Application Server Test Suite: Integration - Basic 7.1.0.Final-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO] --- maven-dependency-plugin:2.3:copy (copy-annotations-endorsed) @
jboss-as-testsuite-integration-basic ---
[INFO] --- maven-resources-plugin:2.5:resources (default-resources) @
jboss-as-testsuite-integration-basic ---
[INFO] --- maven-compiler-plugin:2.3.2:compile (default-compile) @
jboss-as-testsuite-integration-basic ---
[INFO] --- maven-resources-plugin:2.5:copy-resources (build-jbossas.server) @
jboss-as-testsuite-integration-basic ---
[INFO] --- maven-resources-plugin:2.5:copy-resources (ts.copy-jbossas.groups) @
jboss-as-testsuite-integration-basic ---
[INFO] --- maven-resources-plugin:2.5:testResources (default-testResources) @
jboss-as-testsuite-integration-basic ---
[INFO] --- maven-antrun-plugin:1.6:run (prepare-jars-basic-integration.server) @
jboss-as-testsuite-integration-basic ---
[INFO] --- maven-compiler-plugin:2.3.2:testCompile (default-testCompile) @
jboss-as-testsuite-integration-basic ---
[INFO] --- maven-surefire-plugin:2.10:test (basic-integration-default-full.surefire) @
jboss-as-testsuite-integration-basic ---
 T E S T S
               Tests run: 323, Failures: 0, Errors: 4, Skipped: 30
[INFO] ------------------------------------------------------------------------
[INFO] Building JBoss Application Server Test Suite: Integration - Clustering
7.1.0.Final-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO] --- maven-dependency-plugin:2.3:copy (copy-annotations-endorsed) @
jboss-as-testsuite-integration-clust ---
[INFO] --- maven-resources-plugin:2.5:resources (default-resources) @
```

```
jboss-as-testsuite-integration-clust ---
[INFO] --- maven-compiler-plugin:2.3.2:compile (default-compile) @
jboss-as-testsuite-integration-clust ---
[INFO] --- maven-resources-plugin:2.5:copy-resources (build-jbossas.server) @
jboss-as-testsuite-integration-clust ---
[INFO] --- maven-resources-plugin:2.5:copy-resources (ts.copy-jbossas.groups) @
jboss-as-testsuite-integration-clust ---
[INFO] --- maven-resources-plugin:2.5:testResources (default-testResources) @
jboss-as-testsuite-integration-clust ---
[INFO] --- maven-antrun-plugin:1.6:run (build-clustering.server) @
jboss-as-testsuite-integration-clust ---
     [echo] Building config clustering-udp-0
     [echo] Building AS instance "clustering-udp-0" from
/home/ondra/work/AS7/ozizka-git/testsuite/integration/clust/../../../build/target/jboss-as-7.1.0.F
to /home/ondra/work/AS7/ozizka-git/testsuite/integration/clust/target
     [echo] Building config clustering-udp-1
     [echo] Building AS instance "clustering-udp-1" from
/home/ondra/work/AS7/ozizka-git/testsuite/integration/clust/../../../build/target/jboss-as-7.1.0.F
to /home/ondra/work/AS7/ozizka-git/testsuite/integration/clust/target
[INFO] --- maven-compiler-plugin:2.3.2:testCompile (default-testCompile) @
jboss-as-testsuite-integration-clust ---
[INFO] --- maven-surefire-plugin:2.10:test (tests-clustering-multi-node-unmanaged.surefire) @
jboss-as-testsuite-integration-clust ---
 T E S T S
               Tests run: 9, Failures: 0, Errors: 0, Skipped: 0
[INFO] --- maven-surefire-plugin:2.10:test (tests-clustering-single-node.surefire) @
jboss-as-testsuite-integration-clust ---
 T E S T S
               Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO] --- maven-surefire-plugin:2.10:test (tests-clustering-multi-node.surefire) @
jboss-as-testsuite-integration-clust ---
 T E S T S
               Tests run: 8, Failures: 0, Errors: 0, Skipped: 0
[INFO] --- maven-jar-plugin:2.3.1:jar (default-jar) @ jboss-as-testsuite-integration-clust ---
[INFO] Building jar:
/home/ondra/work/AS7/ozizka-git/testsuite/integration/clust/target/jboss-as-testsuite-integration-
--- maven-install-plugin:2.3.1:install (default-install) @ jboss-as-testsuite-integration-clust
---
[INFO] ------------------------------------------------------------------------
[INFO] Building JBoss Application Server Test Suite: Integration - IIOP 7.1.0.Final-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO] --- maven-dependency-plugin:2.3:copy (copy-annotations-endorsed) @
jboss-as-testsuite-integration-iiop ---
[INFO] --- maven-resources-plugin:2.5:resources (default-resources) @
jboss-as-testsuite-integration-iiop ---
[INFO] --- maven-compiler-plugin:2.3.2:compile (default-compile) @
jboss-as-testsuite-integration-iiop ---
[INFO] --- maven-resources-plugin:2.5:copy-resources (build-jbossas.server) @
jboss-as-testsuite-integration-iiop ---
[INFO] --- maven-resources-plugin:2.5:copy-resources (ts.copy-jbossas.groups) @
jboss-as-testsuite-integration-iiop ---
[INFO] --- maven-resources-plugin:2.5:testResources (default-testResources) @
jboss-as-testsuite-integration-iiop ---
[INFO] --- maven-antrun-plugin:1.6:run (build-clustering.server) @
jboss-as-testsuite-integration-iiop ---
     [echo] Building config iiop-client
     [echo] Building AS instance "iiop-client" from
/home/ondra/work/AS7/ozizka-git/testsuite/integration/iiop/../../../build/target/jboss-as-7.1.0.Fi
```

```
to /home/ondra/work/AS7/ozizka-git/testsuite/integration/iiop/target
     [echo] Building config iiop-server
     [echo] Building AS instance "iiop-server" from
/home/ondra/work/AS7/ozizka-git/testsuite/integration/iiop/../../../build/target/jboss-as-7.1.0.Fi
to /home/ondra/work/AS7/ozizka-git/testsuite/integration/iiop/target
[INFO] --- maven-compiler-plugin:2.3.2:testCompile (default-testCompile) @
jboss-as-testsuite-integration-iiop ---
[INFO] --- maven-surefire-plugin:2.10:test (tests-iiop-multi-node.surefire) @
jboss-as-testsuite-integration-iiop ---
 T E S T S
               Tests run: 12, Failures: 0, Errors: 0, Skipped: 0
[INFO] --- maven-jar-plugin:2.3.1:jar (default-jar) @ jboss-as-testsuite-integration-iiop ---
[INFO] Building jar:
/home/ondra/work/AS7/ozizka-git/testsuite/integration/iiop/target/jboss-as-testsuite-integration-i
--- maven-install-plugin:2.3.1:install (default-install) @ jboss-as-testsuite-integration-iiop
---
[INFO] ------------------------------------------------------------------------
[INFO] Building JBoss Application Server Test Suite: Compatibility Tests 7.1.0.Final-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO] --- maven-dependency-plugin:2.3:copy (copy-annotations-endorsed) @
jboss-as-testsuite-integration-compat ---
[INFO] --- maven-resources-plugin:2.5:resources (default-resources) @
jboss-as-testsuite-integration-compat ---
[INFO] --- maven-compiler-plugin:2.3.2:compile (default-compile) @
jboss-as-testsuite-integration-compat ---
[INFO] --- maven-resources-plugin:2.5:copy-resources (build-jbossas.server) @
jboss-as-testsuite-integration-compat ---
[INFO] --- maven-resources-plugin:2.5:testResources (default-testResources) @
jboss-as-testsuite-integration-compat ---
[INFO] --- maven-compiler-plugin:2.3.2:testCompile (default-testCompile) @
jboss-as-testsuite-integration-compat ---
[INFO] --- maven-antrun-plugin:1.6:run (build-jars) @ jboss-as-testsuite-integration-compat ---
[INFO] --- maven-surefire-plugin:2.10:test (default-test) @
jboss-as-testsuite-integration-compat ---
 T E S T S
               Tests run: 7, Failures: 0, Errors: 4, Skipped: 3
[INFO] ------------------------------------------------------------------------
[INFO] Building JBoss Application Server Test Suite: Domain Mode Integration Tests
7.1.0.Final-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO] --- maven-dependency-plugin:2.3:copy (copy-annotations-endorsed) @
jboss-as-testsuite-integration-domain ---
[INFO] --- maven-resources-plugin:2.5:resources (default-resources) @
jboss-as-testsuite-integration-domain ---
[INFO] --- maven-compiler-plugin:2.3.2:compile (default-compile) @
jboss-as-testsuite-integration-domain ---
[INFO] --- maven-resources-plugin:2.5:copy-resources (build-jbossas.server) @
jboss-as-testsuite-integration-domain ---
[INFO] --- maven-resources-plugin:2.5:testResources (default-testResources) @
jboss-as-testsuite-integration-domain ---
[INFO] --- maven-compiler-plugin:2.3.2:testCompile (default-testCompile) @
jboss-as-testsuite-integration-domain ---
[INFO] --- maven-surefire-plugin:2.10:test (default-test) @
jboss-as-testsuite-integration-domain ---
 T E S T S
               Tests run: 89, Failures: 0, Errors: 0, Skipped: 4
[INFO] --- maven-jar-plugin:2.3.1:jar (default-jar) @ jboss-as-testsuite-integration-domain ---
[INFO] Building jar:
```

```
/home/ondra/work/AS7/ozizka-git/testsuite/domain/target/jboss-as-testsuite-integration-domain-7.1.
--- maven-install-plugin:2.3.1:install (default-install) @ jboss-as-testsuite-integration-domain
---
[INFO] ------------------------------------------------------------------------
[INFO] Building JBoss Application Server Test Suite: Benchmark Tests 7.1.0.Final-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO] --- maven-dependency-plugin:2.3:copy (copy-annotations-endorsed) @
jboss-as-testsuite-benchmark ---
[INFO] --- maven-resources-plugin:2.5:resources (default-resources) @
jboss-as-testsuite-benchmark ---
[INFO] --- maven-compiler-plugin:2.3.2:compile (default-compile) @ jboss-as-testsuite-benchmark
---
[INFO] --- maven-resources-plugin:2.5:copy-resources (build-jbossas.server) @
jboss-as-testsuite-benchmark ---
[INFO] --- maven-resources-plugin:2.5:testResources (default-testResources) @
jboss-as-testsuite-benchmark ---
[INFO] --- maven-compiler-plugin:2.3.2:testCompile (default-testCompile) @
jboss-as-testsuite-benchmark ---
[INFO] --- maven-surefire-plugin:2.10:test (default-test) @ jboss-as-testsuite-benchmark ---
 T E S T S
               Tests run: 0, Failures: 0, Errors: 0, Skipped: 0
[INFO] --- maven-jar-plugin:2.3.1:jar (default-jar) @ jboss-as-testsuite-benchmark ---
[INFO] Building jar:
/home/ondra/work/AS7/ozizka-git/testsuite/benchmark/target/jboss-as-testsuite-benchmark-7.1.0.Fina
--- maven-install-plugin:2.3.1:install (default-install) @ jboss-as-testsuite-benchmark ---
[INFO] ------------------------------------------------------------------------
[INFO] Building JBoss Application Server Test Suite: Stress Tests 7.1.0.Final-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO] --- maven-dependency-plugin:2.3:copy (copy-annotations-endorsed) @
jboss-as-testsuite-stress ---
[INFO] --- maven-resources-plugin:2.5:resources (default-resources) @ jboss-as-testsuite-stress
---
[INFO] --- maven-compiler-plugin:2.3.2:compile (default-compile) @ jboss-as-testsuite-stress ---
[INFO] --- maven-resources-plugin:2.5:copy-resources (build-jbossas.server) @
jboss-as-testsuite-stress ---
[INFO] --- maven-resources-plugin:2.5:testResources (default-testResources) @
jboss-as-testsuite-stress ---
[INFO] --- maven-compiler-plugin:2.3.2:testCompile (default-testCompile) @
jboss-as-testsuite-stress ---
[INFO] --- maven-surefire-plugin:2.10:test (default-test) @ jboss-as-testsuite-stress ---
 T E S T S
               Tests run: 0, Failures: 0, Errors: 0, Skipped: 0
[INFO] --- maven-jar-plugin:2.3.1:jar (default-jar) @ jboss-as-testsuite-stress ---
[INFO] Building jar:
/home/ondra/work/AS7/ozizka-git/testsuite/stress/target/jboss-as-testsuite-stress-7.1.0.Final-SNAP
--- maven-install-plugin:2.3.1:install (default-install) @ jboss-as-testsuite-stress ---
[INFO] ------------------------------------------------------------------------
[INFO] ------------------------------------------------------------------------
[INFO] ------------------------------------------------------------------------
[INFO] ------------------------------------------------------------------------
```

13.5 WildFly Testsuite Test Developer Guide

See also: WildFly Integration Testsuite User Guide

13.5.1 requisites

Please be sure to read Pre-requisites - test quality standards and follow those guidelines.

13.5.2 Arquillian container configuration

See AS 7.1 managed container adapter refrence.

13.5.3 ManagementClient and ModelNode usage example

```
final ModelNode operation = new ModelNode();
operation.get(ModelDescriptionConstants.OP).set(ModelDescriptionConstants.READ_RESOURCE_OPERATION);
ModelNode result = managementClient.getControllerClient().execute(operation);
Assert.assertEquals(ModelDescriptionConstants.SUCCESS,
result.get(ModelDescriptionConstants.OUTCOME).asString());
```

ManagementClient can be obtained as described below.

13.5.4 Arquillian features available in tests

@ServerSetup

TBD

```
@ContainerResource private ManagementClient managementClient;
final ModelNode result = managementClient.getControllerClient().execute(operation);
```

TBD

```
@ArquillianResource private ManagementClient managementClient;
ModelControllerClient client = managementClient.getControllerClient();
```

```
@ArquillianResource ContainerController cc;

@Test
public void test() {
    cc.setup("test", ...properties..)
    cc.start("test")
}
```

```
<arquillian>
    <container qualifier="test" mode="manual" />
</arquillian>
```

```
// Targeted containers HTTP context.
@ArquillianResource URL url;
```

```
// Targeted containers HTTP context where servlet is located.
@ArquillianResource(SomeServlet.class) URL url;
```

```
// Targeted containers initial context.
@ArquillianResource InitialContext|Context context;
```

```
// The manual deployer.
@ArquillianResource Deployer deployer;
```

See Arquillian's Resource Injection docs for more info, https://github.com/arquillian/arquillian-examples for examples.

See also Arquillian Reference.

Note to @ServerSetup annotation: It works as expected only on non-manual containers. In case of manual mode containers it calls setup() method after each server start up which is right (or actually before deployment), but the tearDown() method is called only at AfterClass event, i.e. usually after your manual shutdown of the server. Which limits you on the ability to revert some configuration changes on the server and so on. I cloned the annotation and changed it to fit the manual mode, but it is still in my github branch :)

13.5.5 Properties available in tests

Directories

- jbosssa.project.dir - Project's root dir (where ./build.sh is).
- jbossas.ts.dir - Testsuite dir.
- jbossas.ts.integ.dir - Testsuite's integration module dir.
- jboss.dist - Path to AS distribution, either built (build/target/jboss-as-...) or user-provided via -Djboss.dist
- jboss.inst - (Arquillian in-container only) Path to the AS instance in which the test is running (until ARQ-650 is possibly done)

- ~~jboss.home~~ - Deprecated as it's name is unclear and confusing. Use jboss.dist or jboss.inst.

Networking

- node0
- node1
- 230.0.0.4

Time-related coefficients (ratios)

In case some of the following causes timeouts, you may prolong the timeouts by setting value >= 100:

100 = leave as is,
150 = 50 % longer, etc.

- timeout.ratio.gen - General ratio - can be used to adjust all timeouts.When this and specific are defined, both apply.

- timeout.ratio.fs- Filesystem IO
- timeout.ratio.net - Network IO
- timeout.ratio.mem - Memory IO
- timeout.ratio.cpu - Processor
- timeout.ratio.db - Database

Time ratios will soon be provided by `org.jboss.as.test.shared.time.TimeRatio.for*()` methods.

13.5.6 Negative tests

To test invalid deployment handling: @ShouldThrowException

Currently doesn't work due to WFLY-673.

optionally you might be able to catch it using the manual deployer

```
@Deployment(name = "X", managed = false) ...

@Test
public void shouldFail(@ArquillianResource Deployer deployer) throws Exception {
  try {
    deployer.deploy("X")
  }
  catch(Exception e) {
   // do something
  }
}
```

13.5.7 Clustering tests (WFLY-616)

You need to deploy the same thing twice, so two deployment methods that just return the same thing. And then you have tests that run against each.

```
@Deployment(name = "deplA", testable = false)
    @TargetsContainer("serverB")
    public static Archive<?> deployment()

    @Deployment(name = "deplB", testable = false)
    @TargetsContainer("serverA")
    public static Archive<?> deployment(){ ... }

    @Test
    @OperateOnDeployment("deplA")
    public void testA(){ ... }

    @Test
    @OperateOnDeployment("deplA")
    public void testA() {...}
```

13.5.8 How to get the tests to master

- First of all, **be sure to read the "Before you add a test" section**.
- **Fetch** the newest mater: `git fetch upstream  # Provided you have the jbossas/jbossas GitHub repo as a remote called 'upstream'.`
- **Rebase** your branch: git checkout WFLY-1234-your-branch; git rebase upstream/master
- **Run *whole* testsuite** (integration-tests -DallTests). You may use https://jenkins.mw.lab.eng.bos.redhat.com/hudson/job/wildfly-as-testsuite-RHEL-matrix-openJDK7/last .
    - If any tests fail and they do not fail in master, fix it and go back to the "Fetch" step.
- **Push** to a new branch in your GitHub repo: `git push origin WFLY-1234-new-XY-tests`
- **Create a pull-request** on GitHub. Go to your branch and click on "Pull Request".
    - If you have a jira, start the title with it, like - WFLY-1234 New tests for XYZ.
    - If you don't, write some apposite title. In the description, describe in detail what was done and why should it be merged. Keep in mind that the diff will be visible under your description.
- **Keep the branch rebased daily** until it's merged (see the Fetch step). If you don't, you're dramatically decreasing chance to get it merged.
- There's a mailing list, jbossas-pull-requests, which is notified of every pull-request.
- You might have someone with merge privileges to cooperate with you, so they know what you're doing, and expect your pull request.
- When your pull request is reviewed and merged, you'll be notified by mail from GitHub.
- You may also check if it was merged by the following: `git fetch upstream; git cherry <branch>  ## Or` git branch --contains{{<branch> - see}} `here`
- Your commits will appear in master. They will have the same hash as in your branch.
    - You are now safe to delete both your local and remote branches: `git branch -D WFLY-1234-your-branch; git push origin :WFLY-1234-your-branch`

13.5.9 How to Add a Test Case

(Please don't (re)move - this is a landing page from a Jira link.)

Thank you for finding time to contribute to WildFly 8 quality.
Covering corner cases found by community users with tests is very important to increase stability.
If you're providing a test case to support your bug report, it's very likely that your bug will be fixed much sooner.

1) Create a test case.

It's quite easy - a simple use case may even consist of one short .java file.

Check WildFly 8 test suite test cases for examples.

For more information, see WildFly Testsuite Test Developer Guide. Check the requirements for a test to be included in the testsuite.

Ask for help at WildFly 8 forum or at IRC - #wildfly @ FreeNode.

2) Push your test case to GitHub and create a pull request.

For information on how to create a GitHub account and push your code therein, see Hacking on WildFly.

If you're not into Git, send a diff file to JBoss forums, someone might pick it up.

3) Wait for the outcome.

Your test case will be reviewed and eventually added. It may take few days.

When something happens, you'll receive a notification e-mail.

13.5.10 Before you add a test

Every added test, whether ported or new should follow the same guidelines:

- **Verify the test belongs in WildFly 8**

AS6 has a lot of tests for things that are discontinued. For example the legacy JBoss Transaction Manager which was replaced by Arjuna. Also we had tests for SPIs that no longer exist. None of these things should be migrated.

- **Only add CORRECT and UNDERSTANDABLE tests**

If you don't understand what a test is doing (perhaps too complex), or it's going about things in a strange way that might not be correct, THEN DO NOT PORT IT. Instead we should have a simpler, understandable, and correct test. Write a new one, ping the author, or skip it altogether.

- **Do not add duplicate tests**

Always check that the test you are adding doesn't have coverage elsewhere (try using "git grep"). As mentioned above we have some overlap between 6 and 7. The 7 test version will likely be better.

- **Don't name tests after JIRAs**

A JIRA number is useless without an internet connection, and they are hard to read. If I get a test failure thats XDGR-843534578 I have to dig to figure out the context. It's perfectly fine though to link to a JIRA number in the comments of the test. Also the commit log is always available.

- **Tests should contain javadoc that explains what is being tested**

This is especially critical if the test is non-trivial

- **Prefer expanding an EXISTING test over a new test class**

If you are looking at migrating or creating a test with similar functionality to an exiting test, it is better to expand upon the existing one by adding more test methods, rather than creating a whole new test. In general each new test class adds at least 300ms to execution time, so as long as it makes sense it is better to add it to an existing test case.

- **Organize tests by subsystem**

Integration tests should be packaged in subpackages under the relevant subsystem (e.g org.jboss.as.test.integration.ejb.async). When a test impacts multiple subsystems this is a bit of a judgement call, but in general the tests should go into the package of the spec that defines the functionality (e.g. CDI based constructor injection into an EJB, even though this involves CDI and EJB, the CDI spec defines this behaviour)

- **Explain non-obvious spec behavior in comments**

The EE spec is full of odd requirements. If the test is covering behavior that is not obvious then please add something like "Verifies EE X.T.Z - The widget can't have a foobar if it is declared like blah"

- **Put integration test resources in the source directory of the test**

At the moment there is not real organization of these files. It makes sense for most apps to have this separation, however the testsuite is different. e.g. most apps will have a single deployment descriptor of a given type, for the testsuite will have hundreds, and maintaining mirroring package structures is error prone.
This also makes the tests easier to understand, as all the artifacts in the deployment are in one place, and that place tends to be small (only a handful of files).

- **Do not hard-code values likely to need configuration (URLs, ports, ...)**

URLs hardcoded to certain address (localhost) or port (like the default 8080 for web) prevent running the test against different address or with IPv6 adress.
Always use the configurable values provided by Arquillian or as a system property.
If you come across a value which is not configurable but you think it should be, file an WildFly 8 jira issue with component "Test suite".
See @ArquillianResourrce usage example.

- **Follow best committing practices**
  - Only do changes related to the topic of the jira/pull request.
  - Do not clutter your pull request with e.g. reformatting, fixing typos spotted along the way - do another pull request for such.
  - Prefer smaller changes in more pull request over one big pull request which are difficult to merge.
  - Keep the code consistent across commits - e.g. when renaming something, be sure to update all references to it.
  - Describe your commits properly as they will appear in master's linear history.
  - If you're working on a jira issue, include it's ID in the commit message(s).

- **Do not use blind timeouts**

Do not use Thread.sleep() without checking for the actual condition you need to be fulfilled.
You may use active waiting with a timeout, but prefer using timeouts of the API/SPI you test where available.

Make the timeouts configurable: For a group of similar test, use a configurable timeout value with a default if not set.

- **Provide messages in assert*() and fail() calls**

Definitely, it's better to see "File x/y/z.xml not found" instead of:

```
junit.framework.AssertionFailedError
     at junit.framework.Assert.fail(Assert.java:48) [arquillian-service:]
     at junit.framework.Assert.assertTrue(Assert.java:20) [arquillian-service:]
     at junit.framework.Assert.assertTrue(Assert.java:27) [arquillian-service:]
     at
org.jboss.as.test.smoke.embedded.parse.ParseAndMarshalModelsTestCase.getOriginalStandaloneXml(Pars
[bogus.jar:]
```

- **Provide configuration properties hints in exceptions**

If your test uses some configuration property and it fails possibly due to misconfiguration, note the property and it's value in the exception:

```
File jdbcJar = new File( System.getProperty("jbossas.ts.dir", "."),
       "integration/src/test/resources/mysql-connector-java-5.1.15.jar");
    if( !jdbcJar.exists() )
        throw new IllegalStateException("Can't find " + jdbcJar + " using $\{jbossas.ts.dir} ==
" + System.getProperty("jbossas.ts.dir") );
```

- **Clean up**

- - Close sockets, connections, file descriptors;
  - Don't put much data to static fields, or clean them in a finaly {...} block.
  - Don't alter AS config (unless you are absolutely sure that it will reload in a final {...} block or an @After* method)

- **Keep the tests configurable**

  Keep these things in properties, set them at the beginning of the test:
  - Timeouts
  - Paths
  - URLs
  - Numbers (of whatever)

They either will be or already are provided in form of system properties, or a simple testsuite until API (soon to come).

13.5.11 Shared Test Classes and Resources

Among Testsuite Modules

Use the `testsuite/shared` module.

Classes and resources in this module are available in all testsuite modules - i.e. in testsuite/* .

Only use it if necessary - don't put things "for future use" in there.

Don't split packages across modules. Make sure the java package is unique in the WildFly project.

Document your util classes (javadoc) so they can be easily found and reused! A generated list will be put here.

Between Components and Testsuite Modules

To share component's test classes with some module in testsuite, you don't need to split to submodules. You can create a jar with classifier using this:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-jar-plugin</artifactId>
    <executions> <execution>
        <goals> <goal>test-jar</goal> </goals>
    </execution></executions>
</plugin>
```

This creates a jar with classifier "tests", so you can add it as dependency to a testsuite module:

```
<dependency>
   <groupId>org.jboss.as</groupId>
   <artifactId>jboss-as-clustering-common</artifactId>
   <classifier>tests</classifier>
   <version>${project.version}</version>
  <scope>test</scope>
</dependency>
```

14 Quickstarts

WildFly ships with a number of quickstarts that show you how to get started with a variety of technologies in WildFly. You'll find out how to write a web application using the latest Java EE technologies like CDI and JAX-RS. How to write client libraries to talk to WildFly using web services, JMS or EJB. How to set up a distributed transaction, and how to recover from a transaction failure. And much, much more.

14.1 Getting Started

Some of the quickstarts are described in great detail in the Getting Started Developing Applications Guide, which focuses on how get WildFly and Eclipse, with JBoss Tools set up, and how to build web applications.

The other quickstarts are all described in README files and code comments, including what to look out for, how to install and start WildFly, and how to deploy and test the quickstart.

To download the quickstarts, visit https://github.com/wildfly/quickstart.

14.2 Contributing

If you want to contribute to the quickstarts, check out our Contributing a Quickstart page.

14.3 Contributing a Quickstart

Please note: The content for this page has moved.

- JBoss developer information, including the details about the Quickstarts, has moved to the JBoss Developer Framework .
- The list of available quickstarts and download information can be found here: Quickstarts - Get Started
- Information about how to contribute a quickstart can be found in the Contributing Guide located here: Quickstarts - Get Involved with Quickstarts
- The quickstart tool that verifies a quickstart follows the guidelines and uses correct the Maven artifact versions is described here: QS Tools

14.3.1 Maven POM Versions Checklist

Please note: The content for this page has moved.

- JBoss developer information, including the details about the Quickstarts, has moved to the JBoss Developer Framework .
- The list of available quickstarts and download information can be found here: Quickstarts - Get Started
- Information about how to contribute a quickstart can be found in the Contributing Guide located here: Quickstarts - Get Involved with Quickstarts
- The quickstart tool that verifies a quickstart follows the guidelines and uses correct the Maven artifact versions is described here: QS Tools

14.3.2 Writing a quickstart

Please note: The content for this page has moved.

- JBoss developer information, including the details about the Quickstarts, has moved to the JBoss Developer Framework .
- The list of available quickstarts and download information can be found here: Quickstarts - Get Started
- Information about how to contribute a quickstart can be found in the Contributing Guide located here: Quickstarts - Get Involved with Quickstarts
- The quickstart tool that verifies a quickstart follows the guidelines and uses correct the Maven artifact versions is described here: QS Tools

15 WildFly Elytron Security

15.1 About

The WildFly Elytron project is a new security framework brought to WildFly to provide a single unified security framework across the whole of the application server. As a single framework it will be usable both for configuring management access to the server and for applications deployed to the server, it will also be usable across all process types so there will be no need to learn a different security framework for host controllers in a domain compared to configuring a standalone server.

The project covers these main areas: -

- Authentication
- Authorization
- SSL / TLS
- Secure Credential Storage

15.1.1 Authentication

One of the fundamental objectives of the project was to ensure that we can use stronger authentication mechanisms for both HTTP and SASL based authentication, in both cases the new framework also makes it possible to bring in new implementations opening up various integration opportunities with external solutions.

15.1.2 Authorization

The architecture of the project makes a very clear distinction between the raw representation of the identity as returned by a SecurityRealm from the repository of identities and the final representation as a SecurityIdentity after roles have been decoded and mapped and permissions have been mapped.

Custom implementations of the components to perform role decoding and mapping, and permission mapping can be provided allowing for further flexibility beyond the default set of components provided by the project.

15.1.3 SSL / TLS

The project becomes the centralised point within the application server for configuring SSL related resources meaning they can be configured in a central location and referenced by resources across the application server. The centralised configuration also covers advanced options such as configuration of enabled cipher suites and protocols without this information needing to be distributed across the management model.

The SSL / TLS implementation also includes an optimisation where it can be closely tied to authentication allowing for permissions checks to be performed on establishment of a connection before the first request is received and the eager construction of a SecurityIdentity eliminating the need for it to be constructed on a per-request basis.

15.1.4 Secure Credential Storage

The previous vault used for plain text String encryption is replaced with a newly designed credential store. in addition to the protection it offers for the credentials stored within it, the store currently supports storage of clear text credentials.

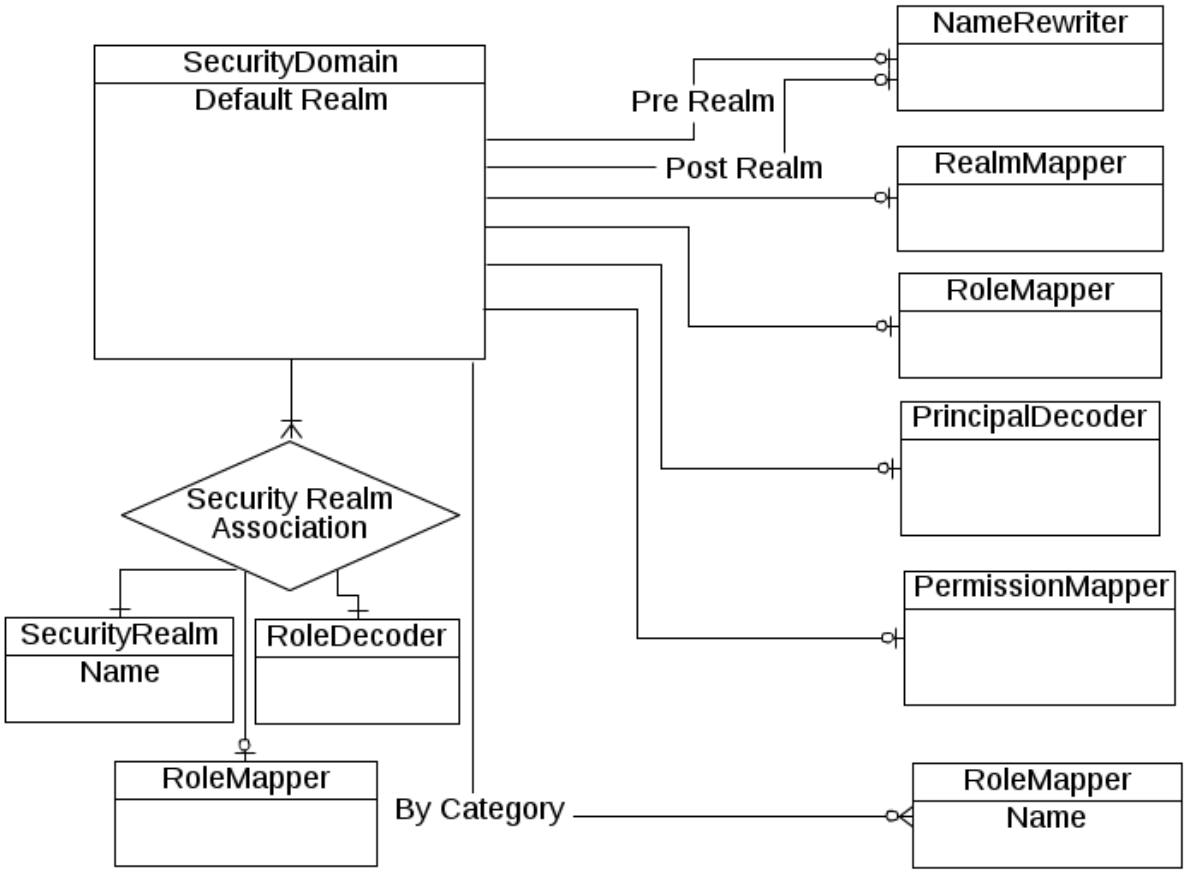
15.2 General Elytron Architecture

The overall architecture for WildFly Elytron is building up a full security policy from assembling smaller components together, by default we include various implementations of the components - in addition to this, custom implementations of many components can be provided in order to provide more specialised implementations.

Within WildFly the different Elytron components are handled as capabilities meaning that different implementations can be mixed and matched, however the different implementations are modelled using distinct resources. This section contains a number of diagrams to show the general relationships between different components to provide a high level view, however the different resource definitions may use different dependencies depending on their purpose.

15.2.1 Security Domains

Within WildFly Elytron a SecurityDomain can be considered as a security policy backed by one or more SecurityRealm instances. Resources that make authorization decisions will be associated with a SecurityDomain, from the SecurityDomain a SecurityIdentity can be obtained which is a representation of the current identity, from this the identities roles and permissions can be checked to make the authorization decision for the resource.



SecurityDomain
The SecurityDomain is the general wrapper around the policy describing a resulting SecurityIdentity and makes use of the following components to define this policy.

- NameRewriter

NameRewriters are used in multiple places within the Elytron configuration, as their name implies, their purpose is to take a name and map it to another representation of the name or perform some normalisation or clean up of the name.

- RealmMapper

As a SecurityDomain is able to reference multiple SecurityRealms the RealmMapper is responsible for identifying which SecurityRealm to use based on the supplied name for authentication.

- SecurityRealm

One more more named SecurityRealms are associated with a SecurityDomain, the SecurityRealms are the access to the underlying repository of identities and are used for obtaining credentials to allow authentication mechanisms to perform verification, for validation of Evidence and for obtaining the raw AuthorizationIdentity performing the authentication.

Some SecurityRealm implementations are also modifiable so expose an API that allows for updates to be made to the repository containing the identities.

- RoleDecoder

Along with the SecurityRealm association is also a reference to a RoleDecoder, the RoleDecoder takes the raw AuthorizationIdentity returned from the SecurityRealm and converts it's attributes into roles.

- RoleMapper

After the roles have been decoded for an identity further mapping can be applied, this could be as simple at normalising the format of the names through to adding or removing specific role names. If a RoleMapper is referenced by the SecurityRealm association that RoleMapper is applied first before applying the RoleMapper associated with the SecurityDomain.

- PrincipalDecoder

A PrincipalDecoder converts from a Principal to a String representation of a name, one example for this is we have an X500PrincipalDecoder which is able to extract an attribute from a distinguished name.

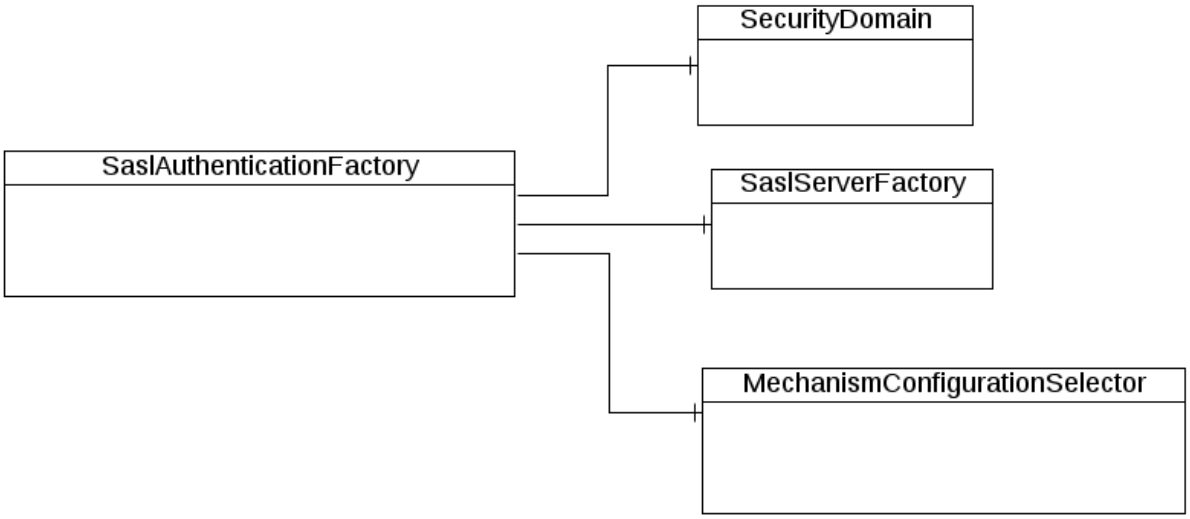
- PermissionMapper

In addition to having roles a SecurityIdentity can also have a set of permissions, the PermissionMapper assigns those permissions to the identity.

Different secured resources can be associated with different SecurityDomains for their authorization decisions, within WildFly Elytron we have the ability to configure inflow between different SecurityDomains. The inflow process means that a SecurityIdentity inflowed into a second SecurityDomain has the mappings of the new SecurityDomain applied to it so although a common identity may be calling different resources each of those resources could have a very different view.

15.2.2 SASL Authentication

The SaslAuthenticationFactory is an authentication policy for authentication using SASL authentication mechanisms, in addition to being a policy it is also a factory for configured authentication mechanisms backed by a SecurityDomain.



SaslAuthenticationFactory
The SaslAuthenticationFactory references the following: -

- SecurityDomain

This is the security domain that any mechanism authentication will be performed against.

- SaslServerFactory

This is the general factory for server side SASL authentication mechanisms.

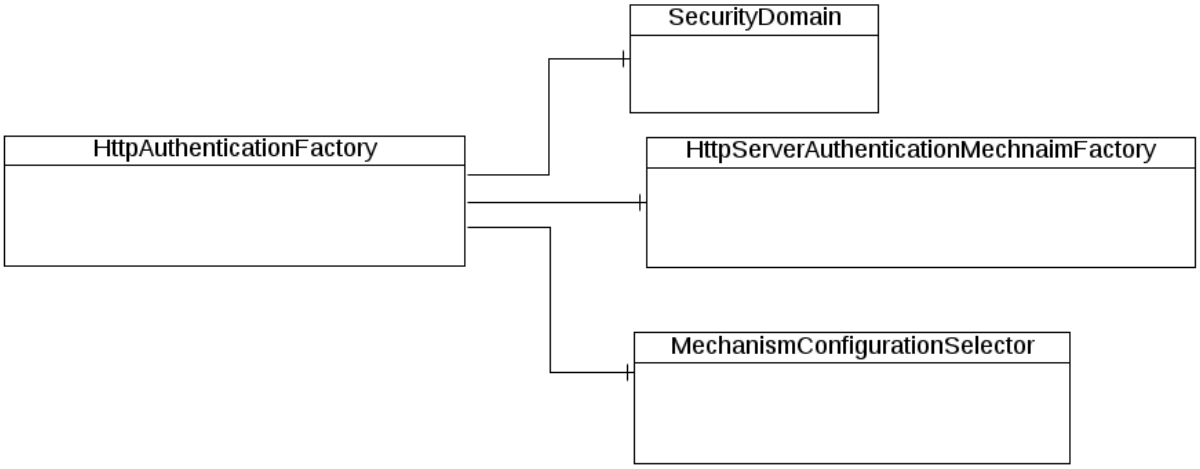
- MechanismConfigurationSelector

Additional configuration can be supplied for the authentication mechanisms, the configuration will be described in more detail later but the purpose of the MechanismConfigurationSelector is to obtain configuration specific to the mechanism selected. This can include information about realm names a mechanism should present to a remote client plus additional NameRewriters and RealmMappers to use during the authentication process.

The reason some of the components referenced by the SecurityDomain are duplicated is so that mechanism specific mappings can be applied.

15.2.3 HTTP Authentication

The HttpAuthenticationFactory is an authentication policy for authentication using HTTP authentication mechanisms, in addition to being a policy it is also a factory for configured authentication mechanisms backed by a SecurityDomain.



HttpAuthenticationFactory
The HttpAuthenticationFactory references the following: -

- SecurityDomain

This is the security domain that any mechanism authentication will be performed against.

- HttpServerAuthenticationMechanismFactory

This is the general factory for server side HTTP authentication mechanisms.

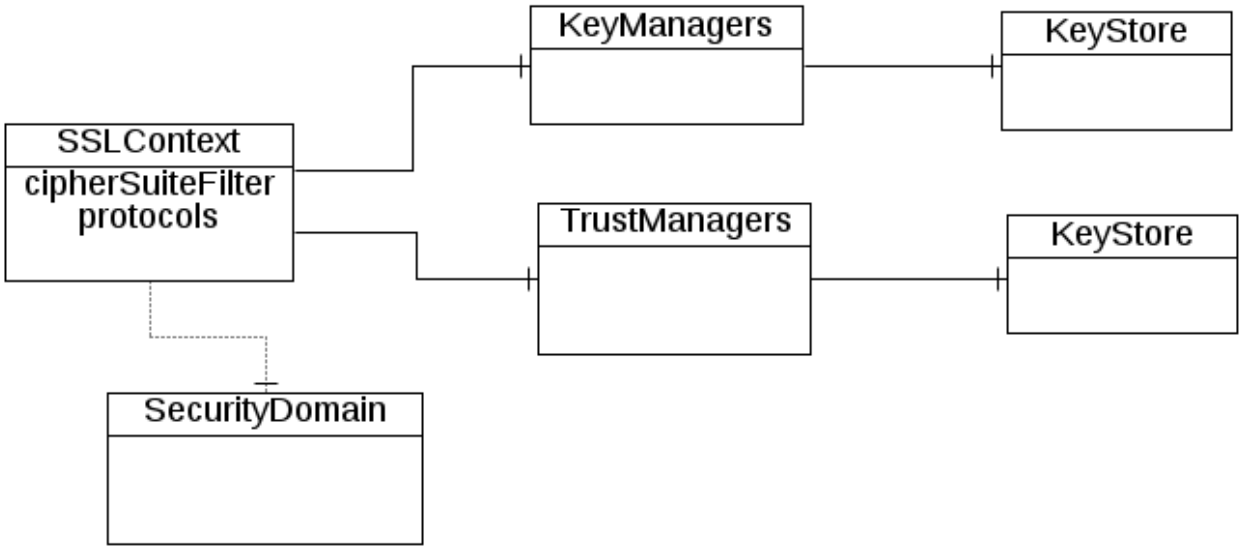
- MechanismConfigurationSelector

Additional configuration can be supplied for the authentication mechanisms, the configuration will be described in more detail later but the purpose of the MechanismConfigurationSelector is to obtain configuration specific to the mechanism selected. This can include information about realm names a mechanism should present to a remote client plus additional NameRewriters and RealmMappers to use during the authentication process.

The reason some of the components referenced by the SecurityDomain are duplicated is so that mechanism specific mappings can be applied.

15.2.4 SSL / TLS

The SSLContext defined within Elytron is a javax.net.ssl.SSLContext meaning it can be used by anything that uses an SSLContext directly.



SSLContext
In addition to the usual configuration for an SSLContext it is possible to configure additional items such as cipher suites and protocols and the SSLContext returned will wrap any engines created to set these values.

The SSLContext within Elytron can also reference the following: -

- KeyManagers

An array of KeyManager instances to be used by the SSLContext, this in turn can reference a KeyStore to load the keys.

- TrustManagers

An array of TrustManager instances to be used by the SSLContext, this in turn can also reference a KeyStore to load the certificates.

- SecurityDomain

This is optional, however if an SSLContext is configured to reference a SecurityDomain then verification of a clients certificate can be performed as an authentication ensuring the appropriate permissions to Logon are assigned before even allowing the connection to be fully opened, additionally the SecurityIdentity can be established at the time the connection is opened and used for any invocations over the connection.

15.3 Elytron Subsystem

WildFly Elytron is a security framework used to unify security across the entire application server. The *elytron* subsystem enables a single point of configuration for securing both applications and the management interfaces. WildFly Elytron also provides a set of APIs and SPIs for providing custom implementations of functionality and integrating with the *elytron* subsystem.

In addition, there are several other important features of the WildFly Elytron:

- Stronger authentication mechanisms for HTTP and SASL authentication.
- Improved architecture that allows for *SecurityIdentities* to be propagated across security domains and transparently transformed ready to be used for authorization. This transformation takes place using configurable role decoders, role mappers, and permission mappers.
- Centralized point for SSL/TLS configuration including cipher suites and protocols.
- SSL/TLS optimizations such as eager *SecureIdentity* construction and closely tying authorization to establishing an SSL/TLS connection. Eager *SecureIdentity* construction eliminates the need for a *SecureIdentity* to be constructed on a per-request basis. Closely tying authentication to establishing an SSL/TLS connection enables permission checks to happen *BEFORE* the first request is received.
- A secure credential store that replaces the previous vault implementation to store clear text credentials.

The new *elytron* subsystem exists in parallel to the legacy *security* subsystem and legacy core management authentication. Both the legacy and Elytron methods may be used for securing the management interfaces as well as providing security for applications.

15.3.1 Get Started using the Elytron Subsystem

To get started using Elytron, refer to these topics:

- Use the default Elytron components for application and management authentication
- Secure an application with a new identity store stored in a filesystem or database.
- Set up one-way SSL/TLS for applications or the management interfaces.
- Set up two-way SSL/TLS for applications or the management interfaces.
- Create a credential store and use it with your SSL/TLS configuration.
- Use certificate-based authentication with applications.
- Override an application's authentication configuration with Elytron authentication.
- Configure Kerberos authentication for applications.
- Secure applications and the management interfaces with an LDAP-based identity store.

15.3.2 Provided components

Wildfly Elytron provides a default set of implementations in the *elytron* subsystem.

Factories

Component	Description
aggregate-http-server-mechanism-factory	An HTTP server factory definition where the HTTP server factory is an aggregation of other HTTP server factories.
aggregate-sasl-server-factory	A SASL server factory definition where the SASL server factory is an aggregation of other SASL server factories.
configurable-http-server-mechanism-factory	A SASL server factory definition where the SASL server factory is an aggregation of other SASL server factories.
configurable-sasl-server-factory	A SASL server factory definition where the SASL server factory is an aggregation of other SASL server factories.
custom-credential-security-factory	A custom credential *SecurityFactory* definition.
http-authentication-factory	Resource containing the association of a security domain with a *HttpServerAuthenticationMechanismFactory*.
kerberos-security-factory	A security factory for obtaining a *GSSCredential* for use during authentication.
mechanism-provider-filtering-sasl-server-factory	A SASL server factory definition that enables filtering by provider where the factory was loaded using a provider.
provider-http-server-mechanism-factory	An HTTP server factory definition where the HTTP server factory is an aggregation of factories from the provider list.
provider-sasl-server-factory	A SASL server factory definition where the SASL server factory is an aggregation of factories from the provider list.
sasl-authentication-factory	Resource containing the association of a security domain with a *SaslServerFactory*.
service-loader-http-server-mechanism-factory	An HTTP server factory definition where the HTTP server factory is an aggregation of factories identified using a *ServiceLoader*
service-loader-sasl-server-factory	A SASL server factory definition where the SASL server factory is an aggregation of factories identified using a *ServiceLoader*

Principal Transformers

Component	Description
aggregate-principal-transformer	A principal transformer definition where the principal transformer is an aggregation of other principal transformers.
chained-principal-transformer	A principal transformer definition where the principal transformer is a chaining of other principal transformers.
constant-principal-transformer	A principal transformer definition where the principal transformer always returns the same constant.
custom-principal-transformer	A custom principal transformer definition.
regex-principal-transformer	A regular expression based principal transformer
regex-validating-principal-transformer	A regular expression based principal transformer which uses the regular expression to validate the name.

Principal Decoders

Component	Description
aggregate-principal-decoder	A principal decoder definition where the principal decoder is an aggregation of other principal decoders.
concatenating-principal-decoder	A principal decoder definition where the principal decoder is a concatenation of other principal decoders.
constant-principal-decoder	Definition of a principal decoder that always returns the same constant.
custom-principal-decoder	Definition of a custom principal decoder.
x500-attribute-principal-decoder	Definition of a X500 attribute based principal decoder.

Realm Mappers

Component	Description
constant-realm-mapper	Definition of a constant realm mapper that always returns the same value.
custom-realm-mapper	Definition of a custom realm mapper
mapped-regex-realm-mapper	Definition of a realm mapper implementation that first uses a regular expression to extract the realm name, this is then converted using the configured mapping of realm names.
simple-regex-realm-mapper	Definition of a simple realm mapper that attempts to extract the realm name using the capture group from the regular expression, if that does not provide a match then the delegate realm mapper is used instead.

Realms

Component	Description
aggregate-realm	A realm definition that is an aggregation of two realms, one for the authentication steps and one for loading the identity for the authorization steps.
caching-realm	A realm definition that enables caching to another security realm. Caching strategy is *Least Recently Used* where least accessed entries are discarded when maximum number of entries is reached.
custom-modifiable-realm	Custom realm configured as being modifiable will be expected to implement the *ModifiableSecurityRealm* interface. By configuring a realm as being modifiable management operations will be made available to manipulate the realm.
custom-realm	A custom realm definitions can implement either the s *SecurityRealm* interface or the *ModifiableSecurityRealm* interface. Regardless of which interface is implemented management operations will not be exposed to manage the realm. However other services that depend on the realm will still be able to perform a type check and cast to gain access to the modification API.
filesystem-realm	A simple security realm definition backed by the filesystem.
identity-realm	A security realm definition where identities are represented in the management model.
jdbc-realm	A security realm definition backed by database using JDBC.
key-store-realm	A security realm definition backed by a keystore.
ldap-realm	A security realm definition backed by LDAP.
properties-realm	A security realm definition backed by properties files.
token-realm	A security realm definition capable of validating and extracting identities from security tokens.
trust-managers	A trust manager definition for creating the *TrustManager* list as used to create an SSL context.

Permission Mappers

Component	Description
custom-permission-mapper	Definition of a custom permission mapper.
logical-permission-mapper	Definition of a logical permission mapper.
simple-permission-mapper	Definition of a simple configured permission mapper.
constant-permission-mapper	Definition of a permission mapper that always returns the same constant.

Role Decoders

Component	Description
custom-role-decoder	Definition of a custom RoleDecoder
simple-role-decoder	Definition of a simple RoleDecoder that takes a single attribute and maps it directly to roles.

Role Mappers

Component	Description
add-prefix-role-mapper	A role mapper definition for a role mapper that adds a prefix to each provided.
add-suffix-role-mapper	A role mapper definition for a role mapper that adds a suffix to each provided.
constant-role-mapper	A role mapper definition where a constant set of roles is always returned.
aggregate-role-mapper	A role mapper definition where the role mapper is an aggregation of other role mappers.
logical-role-mapper	A role mapper definition for a role mapper that performs a logical operation using two referenced role mappers.
custom-role-mapper	Definition of a custom role mapper

SSL Components

Component	Description
client-ssl-context	An SSLContext for use on the client side of a connection.
filtering-key-store	A filtering keystore definition, which provides a keystore by filtering a *key-store*.
key-managers	A key manager definition for creating the key manager list as used to create an SSL context.
key-store	A keystore definition.
ldap-key-store	An LDAP keystore definition, which loads a keystore from an LDAP server.
server-ssl-context	An SSL context for use on the server side of a connection.

Other

Component	Description
aggregate-providers	An aggregation of two or more *Provider[]* resources.
authentication-configuration	An individual authentication configuration definition, which is used by clients deployed to Wildfly and other resources for authenticating when making a remote connection.
authentication-context	An individual authentication context definition, which is used to supply an *ssl-context* and *authentication-configuration* when clients deployed to Wildfly and other resources make a remoting connection.
credential-store	Credential store to keep alias for sensitive information such as passwords for external services.
dir-context	The configuration to connect to a directory (LDAP) server.
provider-loader	A definition for a provider loader.
security-domain	A security domain definition.
security-property	A definition of a security property to be set.

15.3.3 Out of the Box Configuration

WildFly provides a set of components configured by default. While these components are ready to use, the legacy *security* subsystem and legacy core management authentication is still used by default. To configure WildFly to use the these configured components as well as create new ones, see the Using the Elytron Subsystem section.

Default Component	Description
ApplicationDomain	The *ApplicationDomain* security domain uses *ApplicationRealm* and *groups-to-roles* for authentication. I also uses *default-permission-mapper* to assign the login permission.
ManagementDomain	The *ManagementDomain* security domain uses two secur realms for authentication: *ManagementRealm* with *groups-to-roles* and *local* with *super-user-mapper*. It also u *default-permission-mapper* to assign the login permission.
local (security realm)	The *local* security realm does no authentication and sets t identity of principals to *$local*

ApplicationRealm	The *ApplicationRealm* security realm is a properties realm authenticates principals using *application-users.properties* assigns roles using *application-roles.properties*. These file are located under *jboss.server.config.dir*, which by default maps to *EAP_HOME/standalone/configuration*. They are a the same files used by the legacy security default configuration.
ManagementRealm	The *ManagementRealm* security realm is a properties rea that authenticates principals using *mgmt-users.properties* assigns roles using *mgmt-groups.properties*. These files a located under *jboss.server.config.dir*, which by default, ma to *EAP_HOME/standalone/configuration*. They are also th same files used by the legacy security default configuratio
default-permission-mapper	The *default-permission-mapper* mapper is a constant permission mapper that uses *org.wildfly.security.auth.permission.LoginPermission* to as the login permission and *org.wildfly.extension.batch.jberet.deployment.BatchPermis* to assign permission for batch jobs. The batch permission are *start*, *stop*, *restart*, *abandon*, and *read* which aligns wit *javax.batch.operations.JobOperator*.
local (mapper)	The *local* mapper is a constant role mapper that maps to t *local* security realm. This is used to map authentication to *local* security realm.
groups-to-roles	The *groups-to-roles* mapper is a simple-role-decoder that decode the *groups* information of a principal and use it for *role* information.
super-user-mapper	The *super-user-mapper* mapper is a constant role mapper maps the *SuperUser* role to a principal.
management-http-authentication	The *management-http-authentication* http-authentication-factory can be used for doing authentication over http. It uses the *global* provider-http-server-mechanism-factory to filter authentica mechanism and uses *ManagementDomain* for authenticat principals. It accepts the *DIGEST* authentication mechanis and exposes it as *ManagementRealm* to applications.

application-http-authentication	The *application-http-authentication* http-authentication-fac can be used for doing authentication over http. It uses the *global* provider-http-server-mechanism-factory to filter authentication mechanism and uses *ApplicationDomain* fo authenticating principals. It accepts *BASIC* and *FORM* authentication mechanisms and exposes *BASIC* as *Application Realm* to applications.
global (provider-http-server-mechanism-factory)	This is the HTTP server factory mechanism definition used list the provided authentication mechanisms when creating http authentication factory.
management-sasl-authentication	The *management-sasl-authentication* sasl-authentication-factory can be used for authentication using SASL. It uses the *configured* sasl-server-factory to f authentication mechanisms, which also uses the *global* provider-sasl-server-factory to filter by provider names. *management-sasl-authentication* uses the *ManagementDomain* security domain for authentication of principals. It also maps authentication using *JBOSS-LOCAL-USER* mechanisms using the *local* realm mapper and authentication using *DIGEST-MD5* to *ManagementRealm*.
application-sasl-authentication	The *application-sasl-authentication* sasl-authentication-fac can be used for authentication using SASL. It uses the *configured* sasl-server-factory to filter authentication mechanisms, which also uses the *global* provider-sasl-server-factory to filter by provider names. *application-sasl-authentication* uses the *ApplicationDomai* security domain for authentication of principals.
global (provider-sasl-server-factory)	This is the SASL server factory definition used to create S authentication factories.
elytron (mechanism-provider-filtering-sasl-server-factor)	This is used to filter which *sasl-authentication-factory* is us based on the provider. In this case, *elytron* will match on t *WildFlyElytron* provider name.
configured (configurable-sasl-server-factory)	This is used to filter *sasl-authentication-factory* is used bas on the mechanism name. In this case, *configured* will mat on *JBOSS-LOCAL-USER* and *DIGEST-MD5*. It also sets t *wildfly.sasl.local-user.default-user* to *$local*.
combined-providers	Is an aggregate provider that aggreates the *elytron* and *openssl* provider loaders.
elytron	A provider loader

openssl	A provider loader

Default WildFly Configuration

```
/subsystem=elytron:read-resource(recursive=true)
{
    "outcome" => "success",
    "result" => {
        "default-authentication-context" => undefined,
        "final-providers" => undefined,
        "initial-providers" => "combined-providers",
        "add-prefix-role-mapper" => undefined,
        "add-suffix-role-mapper" => undefined,
        "aggregate-http-server-mechanism-factory" => undefined,
        "aggregate-principal-decoder" => undefined,
        "aggregate-principal-transformer" => undefined,
        "aggregate-providers" => {"combined-providers" => {"providers" => [
            "elytron",
            "openssl"
        ]}},
        "aggregate-realm" => undefined,
        "aggregate-role-mapper" => undefined,
        "aggregate-sasl-server-factory" => undefined,
        "authentication-configuration" => undefined,
        "authentication-context" => undefined,
        "caching-realm" => undefined,
        "chained-principal-transformer" => undefined,
        "client-ssl-context" => undefined,
        "concatenating-principal-decoder" => undefined,
        "configurable-http-server-mechanism-factory" => undefined,
        "configurable-sasl-server-factory" => {"configured" => {
            "filters" => [
                {"pattern-filter" => "JBOSS-LOCAL-USER"},
                {"pattern-filter" => "DIGEST-MD5"}
            ],
            "properties" => {"wildfly.sasl.local-user.default-user" => "$local"},
            "protocol" => undefined,
            "sasl-server-factory" => "elytron",
            "server-name" => undefined
        }},
        "constant-permission-mapper" => {"default-permission-mapper" => {"permissions" => [
            {"class-name" => "org.wildfly.security.auth.permission.LoginPermission"},
            {
                "class-name" => "org.wildfly.extension.batch.jberet.deployment.BatchPermission",
                "module" => "org.wildfly.extension.batch.jberet",
                "target-name" => "*"
            }
        ]}},
        "constant-principal-decoder" => undefined,
        "constant-principal-transformer" => undefined,
        "constant-realm-mapper" => {"local" => {"realm-name" => "local"}},
        "constant-role-mapper" => {"super-user-mapper" => {"roles" => ["SuperUser"]}},
        "credential-store" => undefined,
        "custom-credential-security-factory" => undefined,
        "custom-modifiable-realm" => undefined,
```

```
        "custom-permission-mapper" => undefined,
        "custom-principal-decoder" => undefined,
        "custom-principal-transformer" => undefined,
        "custom-realm" => undefined,
        "custom-realm-mapper" => undefined,
        "custom-role-decoder" => undefined,
        "custom-role-mapper" => undefined,
        "dir-context" => undefined,
        "filesystem-realm" => undefined,
        "filtering-key-store" => undefined,
        "http-authentication-factory" => {
            "management-http-authentication" => {
                "http-server-mechanism-factory" => "global",
                "mechanism-configurations" => [{
                    "mechanism-name" => "DIGEST",
                    "mechanism-realm-configurations" => [{"realm-name" => "ManagementRealm"}]
                }],
                "security-domain" => "ManagementDomain"
            },
            "application-http-authentication" => {
                "http-server-mechanism-factory" => "global",
                "mechanism-configurations" => [
                    {
                        "mechanism-name" => "BASIC",
                        "mechanism-realm-configurations" => [{"realm-name" => "Application
Realm"}]
                    },
                    {"mechanism-name" => "FORM"}
                ],
                "security-domain" => "ApplicationDomain"
            }
        },
        "identity-realm" => {"local" => {
            "attribute-name" => undefined,
            "attribute-values" => undefined,
            "identity" => "$local"
        }},
        "jdbc-realm" => undefined,
        "kerberos-security-factory" => undefined,
        "key-managers" => undefined,
        "key-store" => undefined,
        "key-store-realm" => undefined,
        "ldap-key-store" => undefined,
        "ldap-realm" => undefined,
        "logical-permission-mapper" => undefined,
        "logical-role-mapper" => undefined,
        "mapped-regex-realm-mapper" => undefined,
        "mechanism-provider-filtering-sasl-server-factory" => {"elytron" => {
            "enabling" => true,
            "filters" => [{"provider-name" => "WildFlyElytron"}],
            "sasl-server-factory" => "global"
        }},
        "properties-realm" => {
            "ApplicationRealm" => {
                "groups-attribute" => "groups",
                "groups-properties" => {
                    "path" => "application-roles.properties",
                    "relative-to" => "jboss.server.config.dir"
```

```
                },
                "users-properties" => {
                    "path" => "application-users.properties",
                    "relative-to" => "jboss.server.config.dir",
                    "digest-realm-name" => "ApplicationRealm"
                }
            },
            "ManagementRealm" => {
                "groups-attribute" => "groups",
                "groups-properties" => {
                    "path" => "mgmt-groups.properties",
                    "relative-to" => "jboss.server.config.dir"
                },
                "users-properties" => {
                    "path" => "mgmt-users.properties",
                    "relative-to" => "jboss.server.config.dir",
                    "digest-realm-name" => "ManagementRealm"
                }
            }
        },
        "provider-http-server-mechanism-factory" => {"global" => {"providers" => undefined}},
        "provider-loader" => {
            "elytron" => {
                "class-names" => undefined,
                "configuration" => undefined,
                "module" => "org.wildfly.security.elytron",
                "path" => undefined,
                "relative-to" => undefined
            },
            "openssl" => {
                "class-names" => undefined,
                "configuration" => undefined,
                "module" => "org.wildfly.openssl",
                "path" => undefined,
                "relative-to" => undefined
            }
        },
        "provider-sasl-server-factory" => {"global" => {"providers" => undefined}},
        "regex-principal-transformer" => undefined,
        "regex-validating-principal-transformer" => undefined,
        "sasl-authentication-factory" => {
            "management-sasl-authentication" => {
                "mechanism-configurations" => [
                    {
                        "mechanism-name" => "JBOSS-LOCAL-USER",
                        "realm-mapper" => "local"
                    },
                    {
                        "mechanism-name" => "DIGEST-MD5",
                        "mechanism-realm-configurations" => [{"realm-name" =>
"ManagementRealm"}]
                    }
                ],
                "sasl-server-factory" => "configured",
                "security-domain" => "ManagementDomain"
            },
            "application-sasl-authentication" => {
                "mechanism-configurations" => undefined,
```

```
                "sasl-server-factory" => "configured",
                "security-domain" => "ApplicationDomain"
            }
        },
        "security-domain" => {
            "ApplicationDomain" => {
                "default-realm" => "ApplicationRealm",
                "permission-mapper" => "default-permission-mapper",
                "post-realm-principal-transformer" => undefined,
                "pre-realm-principal-transformer" => undefined,
                "principal-decoder" => undefined,
                "realm-mapper" => undefined,
                "realms" => [{
                    "realm" => "ApplicationRealm",
                    "role-decoder" => "groups-to-roles"
                }],
                "role-mapper" => undefined,
                "trusted-security-domains" => undefined
            },
            "ManagementDomain" => {
                "default-realm" => "ManagementRealm",
                "permission-mapper" => "default-permission-mapper",
                "post-realm-principal-transformer" => undefined,
                "pre-realm-principal-transformer" => undefined,
                "principal-decoder" => undefined,
                "realm-mapper" => undefined,
                "realms" => [
                    {
                        "realm" => "ManagementRealm",
                        "role-decoder" => "groups-to-roles"
                    },
                    {
                        "realm" => "local",
                        "role-mapper" => "super-user-mapper"
                    }
                ],
                "role-mapper" => undefined,
                "trusted-security-domains" => undefined
            }
        },
        "security-property" => undefined,
        "server-ssl-context" => undefined,
        "service-loader-http-server-mechanism-factory" => undefined,
        "service-loader-sasl-server-factory" => undefined,
        "simple-permission-mapper" => undefined,
        "simple-regex-realm-mapper" => undefined,
        "simple-role-decoder" => {"groups-to-roles" => {"attribute" => "groups"}},
        "token-realm" => undefined,
        "trust-managers" => undefined,
        "x500-attribute-principal-decoder" => undefined
    }
}
```

15.3.4 Default Application Authentication Configuration

By default, applications are secured using legacy security domains. Applications must specify a security domain in their *web.xml* as well as the authentication method. If no security domain is specified by the application, WildFly will use the provided *other* legacy security domain.

Update WildFly to Use the Default Elytron Components for Application Authentication

```
/subsystem=undertow/application-security-domain=exampleApplicationDomain:add(http-authentication-f
```

Default Elytron Application HTTP Authentication Configuration

By default, the *application-http-authentication* http-authentication-factory is provided for application http authentication.

```
/subsystem=elytron/http-authentication-factory=application-http-authentication:read-resource()
{
    "outcome" => "success",
    "result" => {
        "http-server-mechanism-factory" => "global",
        "mechanism-configurations" => [
            {
                "mechanism-name" => "BASIC",
                "mechanism-realm-configurations" => [{"realm-name" => "Application Realm"}]
            },
            {"mechanism-name" => "FORM"}
        ],
        "security-domain" => "ApplicationDomain"
    }
}
```

The *application-http-authentication* http-authentication-factory is configured to use the *ApplicationDomain* security domain.

```
/subsystem=elytron/security-domain=ApplicationDomain:read-resource()
{
    "outcome" => "success",
    "result" => {
        "default-realm" => "ApplicationRealm",
        "permission-mapper" => "default-permission-mapper",
        "post-realm-principal-transformer" => undefined,
        "pre-realm-principal-transformer" => undefined,
        "principal-decoder" => undefined,
        "realm-mapper" => undefined,
        "realms" => [{
            "realm" => "ApplicationRealm",
            "role-decoder" => "groups-to-roles"
        }],
        "role-mapper" => undefined,
        "trusted-security-domains" => undefined
    }
}
```

The *ApplicationDomain* security domain is backed by the *ApplicationRealm* Elytron security realm, which is a properties-based realm.

```
/subsystem=elytron/properties-realm=ApplicationRealm:read-resource()
{
    "outcome" => "success",
    "result" => {
        "groups-attribute" => "groups",
        "groups-properties" => {
            "path" => "application-roles.properties",
            "relative-to" => "jboss.server.config.dir"
        },
        "users-properties" => {
            "path" => "application-users.properties",
            "relative-to" => "jboss.server.config.dir",
            "digest-realm-name" => "ApplicationRealm"
        }
    }
}
```

15.3.5 Default Management Authentication Configuration

By default, the WildFly management interfaces are secured by the legacy core management authentication.

Default Configuration

```
/core-service=management/management-interface=http-interface:read-resource()
{
    "outcome" => "success",
    "result" => {
        "allowed-origins" => undefined,
        "console-enabled" => true,
        "http-authentication-factory" => undefined,
        "http-upgrade" => {"enabled" => true},
        "http-upgrade-enabled" => true,
        "sasl-protocol" => "remote",
        "secure-socket-binding" => undefined,
        "security-realm" => "ManagementRealm",
        "server-name" => undefined,
        "socket-binding" => "management-http",
        "ssl-context" => undefined
    }
```

WildFly does provide *management-http-authentication* and *management-sasl-authentication* in the *elytron* subsystem for securing the management interfaces as well.

Update WildFly to Use the Default Elytron Components for Management Authentication

Set http-authentication-factory to use management-http-authentication

```
/core-service=management/management-interface=http-interface:write-attribute( \
  name=http-authentication-factory, \
  value=management-http-authentication \
)
```

Set sasl-authentication-factory to use management-sasl-authentication

```
/core-service=management/management-interface=http-interface:write-attribute( \
  name=http-upgrade.sasl-authentication-factory, \
  value=management-sasl-authentication \
)
```

Undefine security-realm

```
/core-service=management/management-interface=http-interface:undefine-attribute(name=security-realm
```

Reload WildFly for the changes to take affect.

```
reload
```

The management interfaces are now secured using the default components provided by the 'elytron' subsystem.

Default Elytron Management HTTP Authentication Configuration

When you access the management interface over HTTP, for example when using the web-based management console, WildFly will use the *management-http-authentication* http-authentication-factory.

```
/subsystem=elytron/http-authentication-factory=management-http-authentication:read-resource()
{
    "outcome" => "success",
    "result" => {
        "http-server-mechanism-factory" => "global",
        "mechanism-configurations" => [{
            "mechanism-name" => "DIGEST",
            "mechanism-realm-configurations" => [{"realm-name" => "ManagementRealm"}]
        }],
        "security-domain" => "ManagementDomain"
    }
}
```

The *management-http-authentication* http-authentication-factory, is configured to use the *ManagementDomain* security domain.

```
/subsystem=elytron/security-domain=ManagementDomain:read-resource()
{
    "outcome" => "success",
    "result" => {
        "default-realm" => "ManagementRealm",
        "permission-mapper" => "default-permission-mapper",
        "post-realm-principal-transformer" => undefined,
        "pre-realm-principal-transformer" => undefined,
        "principal-decoder" => undefined,
        "realm-mapper" => undefined,
        "realms" => [
            {
                "realm" => "ManagementRealm",
                "role-decoder" => "groups-to-roles"
            },
            {
                "realm" => "local",
                "role-mapper" => "super-user-mapper"
            }
        ],
        "role-mapper" => undefined,
        "trusted-security-domains" => undefined
    }
}
```

The *ManagementDomain* security domain is backed by the *ManagementRealm* Elytron security realm, which is a properties-based realm.

```
/subsystem=elytron/properties-realm=ManagementRealm:read-resource()
{
    "outcome" => "success",
    "result" => {
        "groups-attribute" => "groups",
        "groups-properties" => {
            "path" => "mgmt-groups.properties",
            "relative-to" => "jboss.server.config.dir"
        },
        "plain-text" => false,
        "users-properties" => {
            "path" => "mgmt-users.properties",
            "relative-to" => "jboss.server.config.dir"
        }
    }
}
```

Default Elytron Management CLI Authentication

By default, the management CLI (*jboss-cli.sh*) is configured to connect over *remotehttp*.

Default jboss-cli.xml

```
<jboss-cli xmlns="urn:jboss:cli:3.1">

    <default-protocol use-legacy-override="true">remotehttp</default-protocol>

    <!-- The default controller to connect to when 'connect' command is executed w/o arguments
-->
    <default-controller>
        <protocol>remotehttp</protocol>
        <host>localhost</host>
        <port>9990</port>
    </default-controller>
```

This will establish a connection over HTTP and use HTTP upgrade to change the communication protocol to *native*. The HTTP upgrade connection is secured in the *http-upgrade* section of the *http-interface* using a *sasl-authentication-factory*.

Example Configuration with Default Components

```
/core-service=management/management-interface=http-interface:read-resource()
{
    "outcome" => "success",
    "result" => {
        "allowed-origins" => undefined,
        "console-enabled" => true,
        "http-authentication-factory" => "management-http-authentication",
        "http-upgrade" => {
            "enabled" => true,
            "sasl-authentication-factory" => "management-sasl-authentication"
        },
        "http-upgrade-enabled" => true,
        "sasl-protocol" => "remote",
        "secure-socket-binding" => undefined,
        "security-realm" => undefined,
        "server-name" => undefined,
        "socket-binding" => "management-http",
        "ssl-context" => undefined
    }
}
```

The default sasl-authentication-factory is *management-sasl-authentication*.

```
/subsystem=elytron/sasl-authentication-factory=management-sasl-authentication:read-resource()
{
    "outcome" => "success",
    "result" => {
        "mechanism-configurations" => [
            {
                "mechanism-name" => "JBOSS-LOCAL-USER",
                "realm-mapper" => "local"
            },
            {
                "mechanism-name" => "DIGEST-MD5",
                "mechanism-realm-configurations" => [{"realm-name" => "ManagementRealm"}]
            }
        ],
        "sasl-server-factory" => "configured",
        "security-domain" => "ManagementDomain"
    }
}
```

The *management-sasl-authentication* sasl-authentication-factory specifies *JBOSS-LOCAL-USER* and *DIGEST-MD5* mechanisms.

JBOSS-LOCAL-USER Realm

```
/subsystem=elytron/identity-realm=local:read-resource()
{
    "outcome" => "success",
    "result" => {
        "attribute-name" => undefined,
        "attribute-values" => undefined,
        "identity" => "$local"
    }
}
```

The *local* Elytron security realm is for handling silent authentication for local users.

The *ManagementRealm* Elytron security realm is the same realm used in the *management-http-authentication* http-authentication-factory.

15.3.6 Comparing Legacy Approaches to Elytron Approaches

Legacy Approach	Elytron Approach
UsersRoles Login Module	Configure Authentication with a Properties File-Based Identity Store
Database Login Module	Configure Authentication with a Database Identity Store
Ldap, LdapExtended, AdvancedLdap, AdvancedADLdap Login Modules	Configure Authentication with an LDAP-Based Identity Store
Certificate, Certificate Roles Login Module	Configure Authentication with Certificates
Kerberos, SPNEGO Login Modules	Configure Authentication with a Kerberos-Based Identity Store
Kerberos, SPNEGO Login Modules with Fallback	Configure Authentication with a Form as a Fallback for Kerberos
Vault	Create and Use a Credential Store
Legacy Security Realms	Secure the Management Interfaces with a New Identity Store, Silent Authentication
RBAC	Using RBAC with Elytron
Legacy Security Realms for One-way and Two-way SSL/TLS for Applications	Enable One-way SSL/TLS for Applications, Enable Two-way SSL/TLS in WildFly for Applications
Legacy Security Realms for One-way and Two-way SSL/TLS for Management Interfaces	Enable One-way for the Management Interfaces Using the Elytron Subsystem, Enable Two-way SSL/TLS for the Management Interfaces using the Elytron Subsystem

15.4 Using the Elytron Subsystem

15.4.1 Set Up and Configure Authentication for Applications

Configure Authentication with a Properties File-Based Identity Store

Create properties files:

You need to create two properties files: one that maps user to passwords and another that maps users to roles. Usually these files are located in the *jboss.server.config.dir* directory and follow the naming convention **-users.properties* and **-roles.properties*, but other locations and names may be used. The **-users.properties* file must also contain a reference to the *properties-realm*, which you will create in the next step: *#$REALM_NAME=YOUR_PROPERTIES_REALM_NAME$*

Example user to password file: example-users.properties

```
#$REALM_NAME=examplePropRealm$
user1=password123
user2=password123
```

Example user to roles file: example-roles.properties

```
user1=Admin
user2=Guest
```

Configure a properties-realm in WildFly:

```
/subsystem=elytron/properties-realm=examplePropRealm:add(groups-attribute=groups,groups-properties
```

The name of the *properties-realm* is *examplePropRealm*, which is used in the previous step in the *example-users.properties* file. Also, if your properties files are located outside of *jboss.server.config.dir*, then you need to change the *path* and *relative-to* values appropriately.

Configure a security-domain :

```
/subsystem=elytron/security-domain=exampleSD:add(realms=[{realm=examplePropRealm,role-decoder=grou
```

Configure an http-authentication-factory :

```
/subsystem=elytron/http-authentication-factory=example-http-auth:add(http-server-mechanism-factory
```

This example shows creating an *http-authentication-factory* using *BASIC* authentication, but it could be updated to other mechanisms such as *FORM*.

Configure an application-security-domain in the Undertow subsystem:

```
/subsystem=undertow/application-security-domain=exampleApplicationDomain:add(http-authentication-f
```

Configure your application's web.xml and jboss-web.xml .

Your application's *web.xml* and *jboss-web.xml* must be updated to use the *application-security-domain* you configured in WildFly. An example of this is available in the Configure Applications to Use Elytron or Legacy Security for Authentication section.

Configure Authentication with a Filesystem-Based Identity Store

Chose a directory for users:

You need a directory where your users will be stored. In this example, we are using a directory called *fs-realm-users* located in *jboss.server.config.dir*.

Configure a filesystem-realm in WildFly:

```
/subsystem=elytron/filesystem-realm=exampleFsRealm:add(path=fs-realm-users,relative-to=jboss.serve
```

If your directory is located outside of *jboss.server.config.dir*, then you need to change the *path* and *relative-to* values appropriately.

Add a user:

When using the *filesystem-realm*, you can add users using the management CLI.

```
/subsystem=elytron/filesystem-realm=exampleFsRealm/identity=user1:add()
/subsystem=elytron/filesystem-realm=exampleFsRealm/identity=user1:set-password(
clear={password="password123"})
/subsystem=elytron/filesystem-realm=exampleFsRealm/identity=user1:add-attribute(name=Roles,
value=["Admin","Guest"])
```

Add a simple-role-decoder :

```
/subsystem=elytron/simple-role-decoder=from-roles-attribute:add(attribute=Roles)
```

This *simple-role-decoder* decodes a principal's roles from the *Roles* attribute. You can change this value if your roles are in a different attribute.

Configure a security-domain :

```
/subsystem=elytron/security-domain=exampleFsSD:add(realms=[{realm=exampleFsRealm,role-decoder=from
```

Configure an http-authentication-factory :

```
/subsystem=elytron/http-authentication-factory=example-fs-http-auth:add(http-server-mechanism-fact
```

This example shows creating an *http-authentication-factory* using *BASIC* authentication, but it could be updated to other mechanisms such as *FORM*.

Configure an application-security-domain in the Undertow subsystem:

```
/subsystem=undertow/application-security-domain=exampleApplicationDomain:add(http-authentication-f
```

Configure your application's web.xml and jboss-web.xml .

Your application's *web.xml* and *jboss-web.xml* must be updated to use the *application-security-domain* you configured in WildFly. An example of this is available in the Configure Applications to Use Elytron or Legacy Security for Authentication section.

Your application is now using a filesystem-based identity store for authentication.

Configure Authentication with a Database Identity Store

Determine your database format for usernames, passwords, and roles:

To set up authentication using a database for an identity store, you need to determine how your usernames, passwords, and roles are stored in that database. In this example, we are using a single table with the following sample data:

username	password	roles
user1	password123	Admin
user2	password123	Guest

Configure a datasource:

To connect to a database from WildFly, you must have the appropriate database driver deployed as well as a datasource configured. This example shows deploying the driver for postgres and configuring a datasource in WildFly:

```
deploy /path/to/postgresql-9.4.1210.jar

data-source add --name=examplePostgresDS --jndi-name=java:jboss/examplePostgresDS
--driver-name=postgresql-9.4.1210.jar
--connection-url=jdbc:postgresql://localhost:5432/postgresdb --user-name=postgresAdmin
--password=mysecretpassword
```

Configure a jdbc-realm in WildFly:

```
/subsystem=elytron/jdbc-realm=exampleDbRealm:add(principal-query=[{sql="SELECT password,roles
FROM wildfly_users WHERE
username=?",data-source=examplePostgresDS,clear-password-mapper={password-index=1},attribute-mappi
```

NOTE: The above example shows how to obtain passwords and roles from a single *principal-query*. You can also create additional *principal-query* with *attribute-mapping* attributes if you require multiple queries to obtain roles or additional authentication or authorization information.

Configure a security-domain :

```
/subsystem=elytron/security-domain=exampleDbSD:add(realms=[{realm=exampleDbRealm,role-decoder=grou
```

Configure an http-authentication-factory :

```
/subsystem=elytron/http-authentication-factory=example-db-http-auth:add(http-server-mechanism-fact
```

This example shows creating an *http-authentication-factory* using *BASIC* authentication, but it could be updated to other mechanisms such as *FORM*.

Configure an application-security-domain in the Undertow subsystem:

```
/subsystem=undertow/application-security-domain=exampleApplicationDomain:add(http-authentication-f
```

Configure your application's web.xml and jboss-web.xml .

Your application's *web.xml* and *jboss-web.xml* must be updated to use the *application-security-domain* you configured in WildFly. An example of this is available in the Configure Applications to Use Elytron or Legacy Security for Authentication section.

Configure Authentication with an LDAP-Based Identity Store

Determine your LDAP format for usernames, passwords, and roles:

To set up authentication using an LDAP server for an identity store, you need to determine how your usernames, passwords, and roles are stored. In this example, we are using the following structure:

```
dn: dc=wildfly,dc=org
dc: wildfly
objectClass: top
objectClass: domain

dn: ou=Users,dc=wildfly,dc=org
objectClass: organizationalUnit
objectClass: top
ou: Users

dn: uid=jsmith,ou=Users,dc=wildfly,dc=org
objectClass: top
objectClass: person
objectClass: inetOrgPerson
cn: John Smith
sn: smith
uid: jsmith
userPassword: password123

dn: ou=Roles,dc=wildfly,dc=org
objectclass: top
objectclass: organizationalUnit
ou: Roles

dn: cn=Admin,ou=Roles,dc=wildfly,dc=org
objectClass: top
objectClass: groupOfNames
cn: Admin
member: uid=jsmith,ou=Users,dc=wildfly,dc=org
```

Configure a dir-context :

To connect to the LDAP server from WildFly, you need to configure a *dir-context* that provides the URL as well as the principal used to connect to the server.

```
/subsystem=elytron/dir-context=exampleDC:add(url="ldap://127.0.0.1:10389",principal="uid=admin,ou=
```

Configure an ldap-realm in WildFly:

```
/subsystem=elytron/ldap-realm=exampleLR:add(dir-context=exampleDC,identity-mapping={search-base-dn
```

Add a simple-role-decoder :

```
/subsystem=elytron/simple-role-decoder=from-roles-attribute:add(attribute=Roles)
```

Configure a security-domain :

```
/subsystem=elytron/security-domain=exampleLdapSD:add(realms=[{realm=exampleLR,role-decoder=from-ro
```

Configure an http-authentication-factory :

```
/subsystem=elytron/http-authentication-factory=example-ldap-http-auth:add(http-server-mechanism-fa
```

This example shows creating an *http-authentication-factory* using *BASIC* authentication, but it could be updated to other mechanisms such as *FORM*.

Configure an application-security-domain in the Undertow subsystem:

```
/subsystem=undertow/application-security-domain=exampleApplicationDomain:add(http-authentication-f
```

Configure your application's web.xml and jboss-web.xml .

Your application's *web.xml* and *jboss-web.xml* must be updated to use the *application-security-domain* you configured in WildFly. An example of this is available in the Configure Applications to Use Elytron or Legacy Security for Authentication section.

IMPORTANT: In cases where you configure an LDAP server in the *elytron* subsystem for authentication and that LDAP server then becomes unreachable, WildFly will return a *500*, or internal server error, error code when attempting authentication using that unreachable LDAP server. This behavior differs from the legacy *security* subsystem, which will return a *401*, or unauthorized, error code under the same conditions.

Configure Authentication with Certificates

IMPORTANT: Before you can set up certificate-based authentication, you must have two-way SSL configured.

Configure a key-store-realm .

```
/subsystem=elytron/key-store-realm=ksRealm:add(key-store=twoWayTS)
```

You must configure this realm with a truststore that contains the client's certificate. The authentication process uses the same certificate presented by the client during the two-way SSL handshake.

Create a Decoder.

You need to create a *x500-attribute-principal-decoder* to decode the principal you get from your certificate. The below example will decode the principal based on the first *CN* value.

```
/subsystem=elytron/x500-attribute-principal-decoder=CNDecoder:add(oid="2.5.4.3",maximum-segments=1
```

For example, if the full *DN* was *CN=client,CN=client-certificate,DC=example,DC=jboss,DC=org*, *CNDecoder* would decode the principal as *client*. This decoded principal is used as the *alias* value to lookup a certificate in the truststore configured in *ksRealm*.

IMPORTANT: The decoded principal *__MUST__* must be the *alias* value you set in your server's truststore for the client's certificate.

Add a constant-role-mapper for assigning roles.

This is example uses a *constant-role-mapper* to assign roles to a principal from *ksRealm* but other approaches may also be used.

```
/subsystem=elytron/constant-role-mapper=constantClientCertRole:add(roles=[Admin,Guest])
```

Configure a security-domain .

```
/subsystem=elytron/security-domain=exampleCertSD:add(realms=[{realm=ksRealm}],default-realm=ksReal
```

Configure an http-authentication-factory .

```
/subsystem=elytron/http-authentication-factory=exampleCertHttpAuth:add(http-server-mechanism-facto
```

Configure an application-security-domain in the Undertow subsystem.

```
/subsystem=undertow/application-security-domain=exampleApplicationDomain:add(http-authentication-f
```

Update server-ssl-context .

```
/subsystem=elytron/server-ssl-context=twoWaySSC:write-attribute(name=security-domain,value=example
value=true)
```

Configure your application's web.xml and jboss-web.xml .

Your application's *web.xml* and *jboss-web.xml* must be updated to use the *application-security-domain* you configured in WildFly. An example of this is available in the Configure Applications to Use Elytron or Legacy Security for Authentication section.

In addition, you need to update your *web.xml* to use *CLIENT-CERT* as its authentication method.

```
<login-config>
  <auth-method>CLIENT-CERT</auth-method>
  <realm-name>exampleApplicationDomain</realm-name>
</login-config>
```

Configure Authentication with a Kerberos-Based Identity Store

IMPORTANT: The following steps assume you have a working KDC and Kerberos domain as well as your client browsers configured.

Configure a kerberos-security-factory .

```
/subsystem=elytron/kerberos-security-factory=krbSF:add(principal="HTTP/host@REALM",path="/path/to/
```

Configure the system properties for Kerberos.

Depending on how your environment is configured, you will need to set some of the system properties below.

System Property	Description
java.security.krb5.kdc	The host name of the KDC.
java.security.krb5.realm	The name of the realm.
java.security.krb5.conf	The path to the configuration *krb5.conf* file.
sun.security.krb5.debug	If *true*, debugging mode will be enabled.

To configure a system property in WildFly:

```
/system-property=java.security.krb5.conf:add(value="/path/to/krb5.conf")
```

Configure an Eltyron security realm for assigning roles.

The the client's Kerberos token will provide the principal, but you need a way to map that principal to a role for your application. There are several ways to accomplish this, but this example creates a *filesystem-realm*, adds a user to the realm that matches the principal from the Kerberos token, and assigns roles to that user.

```
/subsystem=elytron/filesystem-realm=exampleFsRealm:add(path=fs-realm-users,relative-to=jboss.serve
value=["Admin","Guest"])
```

Add a simple-role-decoder .

```
/subsystem=elytron/simple-role-decoder=from-roles-attribute:add(attribute=Roles)
```

This *simple-role-decoder* decodes a principal's roles from the *Roles* attribute. You can change this value if your roles are in a different attribute.

Configure a security-domain .

```
/subsystem=elytron/security-domain=exampleFsSD:add(realms=[{realm=exampleFsRealm,role-decoder=from
```

Configure an http-authentication-factory that uses the kerberos-security-factory .

```
/subsystem=elytron/http-authentication-factory=example-krb-http-auth:add(http-server-mechanism-fac
```

Configure an application-security-domain in the Undertow subsystem:

```
/subsystem=undertow/application-security-domain=exampleApplicationDomain:add(http-authentication-f
```

Configure your application's web.xml , jboss-web.xml and jboss-deployment-structure.xml .

Your application's *web.xml* and *jboss-web.xml* must be updated to use the *application-security-domain* you configured in WildFly. An example of this is available in the Configure Applications to Use Elytron or Legacy Security for Authentication section.

In addition, you need to update your *web.xml* to use *SPNEGO* as its authentication method.

```
<login-config>
  <auth-method>SPNEGO</auth-method>
  <realm-name>exampleApplicationDomain</realm-name>
</login-config>
```

Configure Authentication with a Form as a Fallback for Kerberos

Configure kerberos-based authentication.

Configuring kerberos-based authentication is covered in a previous section.

Add a mechanism for FORM authentication in the http-authentication-factory .

You can use the existing *http-authentication-factory* you configured for kerberos-based authentication and and an additional mechanism for *FORM* authentication.

```
/subsystem=elytron/http-authentication-factory=example-krb-http-auth:list-add(name=mechanism-confi
value={mechanism-name=FORM})
```

Add additional fallback principals.

The existing configuration for kerberos-based authentication should already have a security realm configured for mapping principals from kerberos token to roles for the application. You can add additional users for fallback authentication to that realm. For example if you used a *filesystem-realm*, you can simply create a new user with the appropriate roles:

```
/subsystem=elytron/filesystem-realm=exampleFsRealm/identity=fallbackUser1:add()
/subsystem=elytron/filesystem-realm=exampleFsRealm/identity=fallbackUser1:set-password(clear={pass
value=["Admin","Guest"])
```

Update the web.xml for FORM fallback.

You need to update the *web.xml* to use the value *SPNEGO,FORM* for the *auth-method*, which will use *FORM* as a fallback authentication method if *SPNEGO* fails. You also need to specify the location of your login and error pages.

```
<login-config>
  <auth-method>SPNEGO,FORM</auth-method>
  <realm-name>exampleApplicationDomain</realm-name>
  <form-login-config>
    <form-login-page>/login.jsp</form-login-page>
    <form-error-page>/error.jsp</form-error-page>
  </form-login-config>
</login-config>
```

Configure Applications to Use Elytron or Legacy Security for Authentication

After you have configured the *elytron* or legacy *security* subsystems for authentication, you need to configure your application to use it.

Configure your application's web.xml .

Your application's *web.xml* needs to be configured to use the appropriate authentication method. When using *elytron*, this is defined in the *http-authentication-factory* you created. When using the legacy *security* subsystem, this depends on your login module and the type of authentication you want to configure.

Example *web.xml* with *BASIC* Authentication

```
<web-app>
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>secure</web-resource-name>
      <url-pattern>/secure/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
      <role-name>Admin</role-name>
    </auth-constraint>
  </security-constraint>
  <security-role>
    <description>The role that is required to log in to /secure/*</description>
    <role-name>Admin</role-name>
  </security-role>
  <login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>exampleApplicationDomain</realm-name>
  </login-config>
</web-app>
```

Configure your application to use a security domain.

You can configure your application's *jboss-web.xml* to specify the security domain you want to use for authentication. When using the *elytron* subsystem, this is defined when you created the *application-security-domain*. When using the legacy *security* subsystem, this is the name of the legacy security domain.

Example *jboss-web.xml*

```
<jboss-web>
  <security-domain>exampleApplicationDomain</security-domain>
</jboss-web>
```

Using *jboss-web.xml* allows you to configure the security domain for a single application only. Alternatively, you can specify a default security domain for all applications using the *undertow* subsystem. This allows you to omit using *jboss-web.xml* to configure a security domain for an individual application.

```
/subsystem=undertow:write-attribute(name=default-security-domain,
value="exampleApplicationDomain")
```

IMPORTANT: Setting *default-security-domain* in the *undertow* subsystem will apply to **ALL** applications. If *default-security-domain* is set and an application specifies a security domain in a *jboss-web.xml* file, the configuration in *jboss-web.xml* will override the *default-security-domain* in the *undertow* subsystem.

Using Elytron and Legacy Security Subsystems in Parallel

You can define authentication in both the *elytron* and legacy *security* subsystems and use them in parallel. If you use both *jboss-web.xml* and *default-security-domain* in the *undertow* subsystem, WildFly will first try to match the configured security domain in the *elytron* subsystem. If a match is not found, then WildFly will attempt to match the security domain with one configured in the legacy *security* subsystem. If the *elytron* and legacy *security* subsystem each have a security domain with the same name, the *elytron* security domain is used.

Override an Application's Authentication Configuration

You can override the authentication configuration of an application with one configured in WildFly. To do this, use the *override-deployment-configuration* property in the *application-security-domain* section of the *undertow* subsystem:

```
/subsystem=undertow/application-security-domain=exampleApplicationDomain:write-attribute(name=over
```

For example, an application is configured to use *FORM* authentication with the *exampleApplicationDomain* in its *jboss-web.xml*.

Example jboss-web.xml

```
<login-config>
  <auth-method>FORM</auth-method>
  <realm-name>exampleApplicationDomain</realm-name>
</login-config>
```

By enabling *override-deployment-configuration*, you can create a new *http-authentication-factory* that specifies a different authentication mechanism such as *BASIC*.

Example http-authentication-factory

```
/subsystem=elytron/http-authentication-factory=exampleHttpAuth:read-resource()
{
    "outcome" => "success",
    "result" => {
        "http-server-mechanism-factory" => "global",
        "mechanism-configurations" => [{
            "mechanism-name" => "BASIC",
            "mechanism-realm-configurations" => [{"realm-name" => "exampleApplicationDomain"}]
        }],
        "security-domain" => "exampleSD"
    }
}
```

This will override the authentication mechanism defined in the application's *jboss-web.xml* and attempt to authenticate a user using *BASIC* instead of *FORM*.

Create and Use a Credential Store

Create credential store.

```
/subsystem=elytron/credential-store=exampleCS:add(uri="cr-store://exampleCS?create=true",credentia
```

Add a credential to a credential store.

```
/subsystem=elytron/credential-store=exampleCS/alias=keystorepw:add(secret-value="secret")
```

List all credentials in a credential store.

```
/subsystem=elytron/credential-store=exampleCS:read-children-names(child-type=alias)
{
    "outcome" => "success",
    "result" => ["keystorepw"]
}
```

Remove a credential from a credential store.

```
/subsystem=elytron/credential-store=exampleCS/alias=keystorepw:remove
```

Use a credential store.

```
/subsystem=elytron/key-store=twoWayKS:write-attribute(name=credential-reference,value={store=examp
```

15.4.2 Set up and Configure Authentication for the Management Interfaces

Secure the Management Interfaces with a New Identity Store

Create a security domain and any supporting security realms, decoders, or mappers for your identity store.

This process is covered in a previous section. For example, if you wanted to secure the management interfaces using a filesystem-based identity store, you would follow the steps in Configure Authentication with a Filesystem-Based Identity Store.

Create an http-authentication-factory or sasl-authentication-factory .

Example *http-authentication-factory*

```
/subsystem=elytron/http-authentication-factory=example-http-auth:add(http-server-mechanism-factory
```

Example *sasl-authentication-factory*

```
/subsystem=elytron/sasl-authentication-factory=example-sasl-auth:add(sasl-server-factory=configure
```

Update the management interfaces to use your http-authentication-factory or sasl-authentication-factory .

Example update *http-authentication-factory*

```
/core-service=management/management-interface=http-interface:write-attribute(name=http-authenticat
value=example-http-auth)
{
   "outcome" => "success",
   "response-headers" => {
       "operation-requires-reload" => true,
       "process-state" => "reload-required"
   }
}

reload
```

Example update *sasl-authentication-factory*

```
/core-service=management/management-interface=http-interface:write-attribute(name=http-upgrade.sas
value=example-sasl-auth)
{
   "outcome" => "success",
   "response-headers" => {
       "operation-requires-reload" => true,
       "process-state" => "reload-required"
   }
}

reload
```

Silent Authentication

By default, WildFly provides an authentication mechanism for local users, also know as silent authentication, through the *local* security realm.

Silent authentication must be used via a *sasl-authentication-factory*.

IMPORTANT: When enabling silent authentication, you must ensure the security domain referenced by your *sasl-authentication-factory* references a security realm that contains the *$local* user. By default, WildFly provides the *local* identity realm that provides this user.

Add silent authentication to an existing sasl-authentication-factory .

```
/subsystem=elytron/sasl-authentication-factory=example-sasl-auth:list-add(name=mechanism-configura
value={mechanism-name=JBOSS-LOCAL-USER, realm-mapper=local})

reload
```

Create a new sasl-server-factory with silent authentication.

```
/subsystem=elytron/sasl-authentication-factory=example-sasl-auth:add(sasl-server-factory=configure
realm-mapper=local}])

reload
```

Remove silent authentication from an existing sasl-server-factory :

```
/subsystem=elytron/sasl-authentication-factory=managenet-sasl-authentication:read-resource
{
    "outcome" => "success",
    "result" => {
        "mechanism-configurations" => [
            {
                "mechanism-name" => "JBOSS-LOCAL-USER",
                "realm-mapper" => "local"
            },
            {
                "mechanism-name" => "DIGEST-MD5",
                "mechanism-realm-configurations" => [{"realm-name" => "ManagementRealm"}]
            }
        ],
        "sasl-server-factory" => "configured",
        "security-domain" => "ManagementDomain"
    }
}

/subsystem=elytron/sasl-authentication-factory=temp-sasl-authentication:list-remove(name=mechanism
```

Using RBAC with Elytron

RBAC can be configured to automatically assign or exclude roles for users that are members of groups. This is configured in the *access-control* section of the core management. When the management interfaces are secured with the *elytron* subsystem, and users are assigned groups when they authenticate. You can also configure roles to be assigned to authenticated users in a variety of ways using the *elytron* subsystem, for example using a role mapper or a role decoder.

15.4.3 Configure SSL/TLS

Enable One-way SSL/TLS for Applications

In WildFly, you can use the Elytron subsystem, along with the Undertow subsystem, to enable HTTPS for deployed applications.

Obtain or generate your key store:

Before enabling HTTPS in WildFly, you must obtain or generate the keystore you plan on using. To generate an example keystore:

```
$ keytool -genkeypair -alias localhost -keyalg RSA -keysize 1024 -validity 365 -keystore
/path/to/keystore.jks -dname "CN=localhost" -keypass secret -storepass secret
```

Configure a key-store in WildFly:

```
/subsystem=elytron/key-store=httpsKS:add(path=/path/to/keystore.jks,credential-reference={clear-te
```

The previous command uses an absolute path to the keystore. Alternatively you can use the *relative-to* attribute to specify the base directory variable and *path* specify a relative path.

```
/subsystem=elytron/key-store=httpsKS:add(path=keystore.jks,relative-to=jboss.server.config.dir,cre
```

Configure a key-manager in that references your key-store :

```
/subsystem=elytron/key-manager=httpsKM:add(key-store=httpsKS,credential-reference={clear-text=secr
```

Configure a server-ssl-context in that references your key-manager :

```
/subsystem=elytron/server-ssl-context=httpsSSC:add(key-manager=httpsKM,protocols=["TLSv1.2"])
```

IMPORTANT: You need to determine what SSL/TLS protocols you want to support. The example commands above uses *TLSv1.2*.

Check and see if the https-listener is configured to use a legacy security realm for its SSL configuration:

```
/subsystem=undertow/server=default-server/https-listener=https:read-attribute(name=security-realm)
"outcome" => "success",
    "result" => "ApplicationRealm"
}
```

The above command shows that the *https-listener* is configured to use the *ApplicationRealm* legacy security realm for its SSL configuration. Undertow cannot reference both a legacy security realm and an *ssl-context* in Elytron at the same time so you must remove the reference to the legacy security realm. Also there has to be always configured either *ssl-context* or *security-realm*. Thus when changing between those, you have to use batch operation:

Remove the reference to the legacy security realm and update the *https-listener* to use the *ssl-context* from Elytron :

```
batch
/subsystem=undertow/server=default-server/https-listener=https:undefine-attribute(name=security-re
```

Reload the server:

```
reload
```

HTTPS is now enabled for applications.

Enable Two-way SSL/TLS in WildFly for Applications

In WildFly, you can use the Elytron subsystem, along with the Undertow subsystem, to enable two-way SSL/TLS for deployed applications.

Obtain or generate your keystore:

Before enabling HTTPS in WildFly, you must obtain or generate the keystores, truststores and certificates you plan on using.

Create server and client keystores:

```
$ keytool -genkeypair -alias localhost -keyalg RSA -keysize 1024 -validity 365 -keystore
server.keystore.jks -dname "CN=localhost" -keypass secret -storepass secret

$ keytool -genkeypair -alias client -keyalg RSA -keysize 1024 -validity 365 -keystore
client.keystore.jks -dname "CN=client" -keypass secret -storepass secret
```

Export the server and client certificates:

```
$ keytool -exportcert  -keystore server.keystore.jks -alias localhost -keypass secret -storepass
secret -file server.cer

$ keytool -exportcert  -keystore client.keystore.jks -alias client -keypass secret -storepass
secret -file client.cer
```

Import the sever and client certificates into the opposing truststores:

```
$ keytool -importcert -keystore server.truststore.jks -storepass secret -alias client
-trustcacerts -file client.cer

$ keytool -importcert -keystore client.truststore.jks -storepass secret -alias localhost
-trustcacerts -file server.cer
```

Configure a key-store for server keystore and truststore in WildFly:

```
/subsystem=elytron/key-store=twoWayKS:add(path=/path/to/server.keystore.jks,credential-reference={
```

NOTE

The previous command uses an absolute path to the keystore. Alternatively you can use the *relative-to* attribute to specify the base directory variable and *path* specify a relative path.

```
/subsystem=elytron/key-store=myKS:add(path=keystore.jks,relative-to=jboss.server.config.dir,creden
```

Configure a key-manager in that references your key store key-store :

```
/subsystem=elytron/key-manager=twoWayKM:add(key-store=twoWayKS,credential-reference={clear-text=se
```

Configure a trust-manager in that references your truststore key-store :

```
/subsystem=elytron/trust-manager=twoWayTM:add(key-store=twoWayTS)
```

Configure a server-ssl-context in that references your key-manager , trust-manager , and enables client authentication:

```
/subsystem=elytron/server-ssl-context=twoWaySSC:add(key-manager=twoWayKM,protocols=["TLSv1.2"],tru
```

IMPORTANT
You need to determine what SSL/TLS protocols you want to support. The example commands above uses *TLSv1.2*.

Check and see if the https-listener is configured to use a legacy security realm for its SSL configuration:

```
/subsystem=undertow/server=default-server/https-listener=https:read-attribute(name=security-realm)
"outcome" => "success",
    "result" => "ApplicationRealm"
}
```

The above command shows that the *https-listener* is configured to use the *ApplicationRealm* legacy security realm for its SSL configuration. Undertow cannot reference both a legacy security realm and an *ssl-context* in Elytron at the same time so you must remove the reference to the legacy security realm. Also there has to be always configured either *ssl-context* or *security-realm*. Thus when changing between those, you have to use batch operation:

Remove the reference to the legacy security realm and update the https-listener to use the ssl-context from Elytron:

```
batch
/subsystem=undertow/server=default-server/https-listener=https:undefine-attribute(name=security-re
```

Reload the server

```
reload
```

Configure your client to use the client certificate

You need to configure your client to present the trusted client certificate to the server to complete the two-way SSL/TLS authentication. For example, if using a browser, you need to import the trusted certificate into the browser's truststore.

Two-Way HTTPS is now enabled for applications.

Enable One-way SSL/TLS for the Management Interfaces Using the Elytron Subsystem

Obtain or generate your key store:

Before enabling HTTPS in WildFly, you must obtain or generate the key store you plan on using. To generate an example key store, use the following command.

```
$ keytool -genkeypair -alias localhost -keyalg RSA -keysize 1024 -validity 365 -keystore
keystore.jks -dname "CN=localhost" -keypass secret -storepass secret
```

Create a key-store , key-manager , and server-ssl-context .

```
/subsystem=elytron/key-store=httpsKS:add(path=keystore.jks,relative-to=jboss.server.config.dir,cre
```

IMPORTANT: You need to determine what SSL/TLS protocols you want to support. The example commands above uses *TLSv1.2*.

NOTE: The above command uses *relative-to* to reference the location of the keystore file. Alternatively, you can specify the full path to the keystore in *path* and omit *relative-to*.

Enable HTTPS on the management interface.

```
/core-service=management/management-interface=http-interface:write-attribute(name=ssl-context,
value=httpsSSC)

/core-service=management/management-interface=http-interface:write-attribute(name=secure-socket-bi
value=management-https)
```

Reload the WildFly instance.

```
reload
```

HTTPS is now enabled for the management interfaces.

Enable Two-way SSL/TLS for the Management Interfaces using the Elytron Subsystem

Obtain or generate your key store.

Before enabling HTTPS in WildFly, you must obtain or generate the key stores, trust stores and certificates you plan on using. To generate an example set of key stores, trust stores, and certificates use the following commands.

Generate your server and client key stores.

```
$ keytool -genkeypair -alias localhost -keyalg RSA -keysize 1024 -validity 365 -keystore
server.keystore.jks -dname "CN=localhost" -keypass secret -storepass secret

$ keytool -genkeypair -alias client -keyalg RSA -keysize 1024 -validity 365 -keystore
client.keystore.jks -dname "CN=client" -keypass secret -storepass secret
```

Export your server and client certificates.

```
$ keytool -exportcert  -keystore server.keystore.jks -alias localhost -keypass secret -storepass
secret -file server.cer

$ keytool -exportcert  -keystore client.keystore.jks -alias client -keypass secret -storepass
secret -file client.cer
```

Import the sever and client certificates into the opposing trust stores.

```
$ keytool -importcert -keystore server.truststore.jks -storepass secret -alias client
-trustcacerts -file client.cer

$ keytool -importcert -keystore client.truststore.jks -storepass secret -alias localhost
-trustcacerts -file server.cer
```

Configure key-store , a key-manager , trust-manager , and server-ssl-context for the server key store and trust store.

```
/subsystem=elytron/key-store=twoWayKS:add(path=server.keystore.jks,relative-to=jboss.server.config
```

IMPORTANT: You need to determine what SSL/TLS protocols you want to support. The example commands above uses *TLSv1.2*.

NOTE: The above command uses *relative-to* to reference the location of the keystore file. Alternatively, you can specify the full path to the keystore in *path* and omit *relative-to*.

Enable HTTPS on the management interface.

```
/core-service=management/management-interface=http-interface:write-attribute(name=ssl-context,
value=twoWaySSC)

/core-service=management/management-interface=http-interface:write-attribute(name=secure-socket-bi
value=management-https)
```

Reload the WildFly instance.

```
reload
```

Configure your client to use the client certificate.

You need to configure your client to present the trusted client certificate to the server to complete the two-way SSL/TLS authentication. For example, if using a browser, you need to import the trusted certificate into the browser's trust store.

Two-way SSL/TLS is now enabled for the management interfaces.

Using an ldap-key-store

An *ldap-key-store* allows you to use a keystore stored in an LDAP server. You can use an *ldap-key-store* in same way you can use a *key-store*.

To create and use an *ldap-key-store*:

Configure a dir-context .

To connect to the LDAP server from WildFly, you need to configure a *dir-context* that provides the URL as well as the principal used to connect to the server.

Example dir-context

```
/subsystem=elytron/dir-context=exampleDC:add( \
  url="ldap://127.0.0.1:10389", \
  principal="uid=admin,ou=system", \
  credential-reference={clear-text=secret} \
)
```

Configure an ldap-key-store .

When configure an *ldap-key-store*, you need to specify both the *dir-context* used to connect to the LDAP server as well as how to locate the keystore stored in the LDAP server. At a minimum, this requires you specify a *search-path*.

Example ldap-key-store

```
/subsystem=elytron/ldap-key-store=ldapKS:add( \
  dir-context=exampleDC, \
  search-path="ou=Keystores,dc=wildfly,dc=org" \
)
```

Use the ldap-key-store .

Once you have defined your *ldap-key-store*, you can use it in the same places where a *key-store* could be used. For example, you could use an *ldap-key-store* when configuring HTTPS and Two-Way HTTPS for applications.

Using a filtering-key-store

A *filtering-key-store* allows you to expose a subset of aliases from an existing *key-store*, and use it in the same places you could use a *key-store*. For example, if a keystore contained *alias1*, *alias2*, and *alias3*, but you only wanted to expose *alias1* and *alias3*, a *filtering-key-store* provides you several ways to do that.

To create a *filtering-key-store*:

Configure a key-store .

```
/subsystem=elytron/key-store=myKS:add( \
  path=keystore.jks, \
  relative-to=jboss.server.config.dir, \
  credential-reference={ \
    clear-text=secret \
  }, \
  type=JKS \
)
```

Configure a filtering-key-store .

When you configure a *filtering-key-store*, you specify which *key-store* you want to filter and the *alias-filter* for filtering aliases from the *key-store*. The filter can be specified in one of the following formats:

- *alias1,alias3*, which is a comma-delimited list of aliases to expose.
- *ALL:-alias2*, which exposes all aliases in the keystore except the ones listed.
- *NONE:+alias1:+alias3*, which exposes no aliases in the keystore except the ones listed.

This example uses a comma-delimted list to expose *alias1* and *alias3*.

```
/subsystem=elytron/filtering-key-store=filterKS:add( \
  key-store=myKS, \
  alias-filter="alias1,alias3" \
)
```

Use the filtering-key-store .

Once you have defined your *filtering-key-store*, you can use it in the same places where a *key-store* could be used. For example, you could use a *filtering-key-store* when configuring HTTPS and Two-Way HTTPS for applications.

Reload a Keystore

You can reload a keystore configured in WildFly from the management CLI. This is useful in cases where you have made changes to certificates referenced by a keystore.

To reload a keystore.

```
/subsystem=elytron/key-store=httpsKS:load
```

Check the Content of a Keystore by Alias

If you add a keystore to the *elytron* subsystem using the *key-store* component, you can check the keystore's contents using the *alias* child element and reading its attributes.

For example:

```
/subsystem=elytron/key-store=httpsKS/alias=localhost:read-attribute(name=certificate-chain)
{
    "outcome" => "success",
    "result" => [{
        "type" => "X.509",
        "algorithm" => "RSA",
        "format" => "X.509",
        "public-key" => "30:81:9f:30:0d:06:09:2a:8.....
```

The following attributes can be read:

Attribute	Description
certificate	The certificate associated with the alias. If the alias has a certificate chain this will always be undefined.
certificate-chain	The certificate chain associated with the alias.
creation-date	The creation date of the entry represented by this alias.
entry-type	The type of the entry for this alias. Available types: *PasswordEntry*, *PrivateKeyEntry*, *SecretKeyEntry*, *TrustedCertificateEntry*, and *Other*. Unrecognized types will be reported as *Other*.

Custom Components

When configuring SSL/TLS in the *elytron* subsystem, you can provide and use custom implementations of the following components:

- *key-store*
- *key-manager*
- *trust-manager*
- *client-ssl-context*
- *server-ssl-context*

When creating custom implementations of Elytron components, they must present the appropriate capabilities and requirements.

15.4.4 Configuring the Elytron and Security Subsystems

Enable and Disable the Elytron Subsystem

To add the elytron extension required for the elytron subsystem:

```
/extension=org.wildfly.extension.elytron:add()
```

To enable the Elytron subsystem in WildFly:

```
/subsystem=elytron:add

reload
```

To disable the Elytron subsystem in WildFly:

```
/subsystem=elytron:remove

reload
```

IMPORTANT: Other subsystems within WildFly may have dependencies on the *elytron* subsystem. If these dependencies are not resolved before disabling it, you will see errors when starting WildFly.

Enable and Disable the Security Subsystem

To disable the security subsystem in WildFly:

```
/subsystem=security:remove

reload
```

IMPORTANT: Other subsystems within WildFly may have dependencies on the *security* subsystem. If these dependencies are not resolved before disabling it, you will see errors when starting WildFly.

To enable the security subsystem in WildFly:

```
/subsystem=security:add

reload
```

Use the Elytron and Security Subsystems in Parallel

By default the *elytron* and *security* subsystems will run in parallel if both are enabled. For authentication in applications, you can use the *application-security-domain* property in the *undertow* subsystem to configure a security domain in the *elytron* subsystem.

```
/subsystem=undertow/application-security-domain=exampleApplicationDomain:add(http-authentication-f
```

NOTE: This must match the *security-domain* configured in the *jboss-web.xml* of your application.

If the *application-security-domain* is not set, WildFly will look for a security domain configured in the *security* subsystem that matches the *security-domain* configured in the *jboss-web.xml* of your application.

For enabling HTTPS using a legacy security realm, you can use the *security-realm* attribute in the *https-listener* section of the *undertow* subsystem:

```
/subsystem=undertow/server=default-server/https-listener=https:read-attribute(name=security-realm)
"outcome" => "success",
    "result" => "ApplicationRealm"
}
```

For enabling HTTPS using *elytron*, you need to undefine the *security-realm* attribute and set the *ssl-context* attribute. As there has to be always configured either *ssl-context* or *security-realm* you have to use batch operation when changing between those:

```
batch
/subsystem=undertow/server=default-server/https-listener=https:undefine-attribute(name=security-re
```

15.4.5 Creating Elytron Subsystem Components

Create an Elytron Security Realm

Security realms in the Elytron subsystem, when used in conjunction with security domains, are use for both core management authentication as well as for authentication with applications. Security realms are also specifically typed based on their identity store, for example *jdbc-realm*, *filesystem-realm*, *properties-realm*, etc.

Adding a security realm takes the general form:

```
/subsystem=elytron/type-of-realm=realmName:add(....)
```

Examples of adding specific realms, such as *jdbc-realm*, *filesystem-realm*, and *properties-realm* can be found in previous sections.

Create an Elytron Role Decoder

A role decoder converts attributes from the identity provided by the security realm into roles. Role decoders are also specifically typed based on their functionality, for example *empty-role-decoder*, *simple-role-decoder*, and *custom-role-decoder*.

Adding a role decoder takes the general form:

```
/subsystem=elytron/ROLE-DECODER-TYPE=roleDeoderName:add(....)
```

Create an Elytron Permission Mapper

In addition to roles being assigned to a identity, permissions may also be assigned. A permission mapper assigns permissions to an identity. Permission mappers are also specifically typed based on their functionality, for example *logical-permission-mapper*, *simple-permission-mapper*, and *custom-permission-mapper*.

Adding a permission mapper takes the general form:

```
/subsystem=elytron/simple-permission-mapper=PermissionMapperName:add(...)
```

Create an Elytron Role Mapper

A role mapper maps roles after they have been decoded to other roles. Examples include normalizing role names or adding and removing specific roles from principals after they have been decoded. Role mappers are also specifically typed based on their functionality, for example *add-prefix-role-mapper*, *add-suffix-role-mapper*, and *constant-role-mapper*.

Adding a role mapper takes the general form:

```
/subsystem=elytron/ROLEM-MAPPER-TYPE=roleMapperName:add(...)
```

Create an Elytron Security Domain

Security domains in the Elytron subsystem, when used in conjunction with security realms, are use for both core management authentication as well as for authentication with applications.

Adding a security domain takes the general form:

```
/subsystem=elytron/security-domain=domainName:add(realms=[{realm=realmName,role-decoder=roleDecode
```

Create an Elytron Authentication Factory

An authentication factory is an authentication policy used for specific authentication mechanisms. Authenticaion factories are specifically based on the authentication mechanism, for example *http-authentication-factory* and *sasl-authentication-factory* and *kerberos-security-factory*.

Adding an authentication factory takes the general form:

```
/subsystem=elytron/AUTH-FACTORY-TYPE=authFactoryName:add(....)
```

Create an Elytron Policy Provider

Elytron subsystem provides a specific resource definition that can be used to configure a default Java Policy provider. The subsystem allows you to define multiple policy providers but select a single one as the default:

```
/subsystem=elytron/policy=policy-provider-a:add(custom-policy=\[{name=policy-provider-a,
class-name=MyPolicyProviderA, module=x.y.z}\])
```

```
```

15.5 Using Elytron within WildFly

15.5.1 Using the Out of the Box Elytron Components

Securing Management Interfaces

You can find more details on the enabling WildFly to use the out of the box Elytron components for securing the management interfaces in the Default Management Authentication Configuration section.

Securing Applications

The *elytron* subsystem provides *application-http-authentication* by default which can be used to secure applications. For more details on how *application-http-authentication* is configured, see the Out of the Box Configuration section.

To configure applications to use *application-http-authentication*, see Configure Applications to Use Elytron or Legacy Security for Authentication. You can also override the default behavior of all applications using the steps in Override an Application's Authentication Configuration.

Using SSL/TLS

WildFly does provide a default one-way SSL/TLS configuration using the legacy core management authentication but does not provide one in the *elytron* subsystem. You can find more details on configuring SSL/TLS using the *elytron* subsystem for both the management interfaces as well as for applications in Configure SSL/TLS

Using Elytron with Other Subsystems

In addition to securing applications and management interfaces, Elytron also integrates with other subsystems in WildFly.

Subsystem	Details
batch-jberet	You can configure the *batch-jberet* to run batch jobs using an Elytron security domain.
datasources	You can use a credential store or an Elytron security domain to provide authentication information in a datasource definition.
messaging-activemq	You can secure remote connections to the remote connections used by the *messaging-activemq* subsystem.
iiop-openjdk	You can use the *elytron* subsystem to configure SSL/TLS between clients and servers using the *iiop-openjdk* subsystem.
mail	You can use a credential store to provide authentication information in a server definition in the *mail* subsystem.
undertow	You can use the *elytron* subsystem to configure both SSL/TLS and application authentication.

15.5.2 Undertow Subsystem

15.5.3 EJB Subsystem

Configuration can be added to the EJB subsystem to map a security domain name referenced in a deployment to an Elytron security domain:

```
/subsystem=ejb3/application-security-domain=MyAppSecurity:add(security-domain=ApplicationDomain)
```

Which results in:

```
<subsystem xmlns="urn:jboss:domain:ejb3:5.0">
...
    <application-security-domains>
        <application-security-domain name="MyAppSecurity" security-domain="ApplicationDomain"/>
    </application-security-domains>
...
</subsystem>
```

Note: If the deployment was already deployed at this point the application server should be reloaded or the deployment redeployed for the application security domain mapping to take effect.

An `application-security-domain` has two main attributes:

- name - the name of the security domain as specified in a deployment
- security-domain - a reference to the Elytron security domain that should be used

When an application security domain mapping is configured for a bean in a deployment, this indicates that security should be handled by Elytron.

15.5.4 WebServices Subsystem

There is adapter in webservices subsystem to make authentication works for elytron security domain automatically. Like configure with legacy security domain, you can configure elytron security domain in deployment descriptor or annotation to secure webservice endpoint.

15.5.5 Legacy Security Subsystem

As previously described, Elytron based security is configured by chaining together different capability references to form a complete security policy. To allow an incremental migration from the legacy Security subsystem some of the major components of this subsystem can be mapped to Elytron capabilities and used within an Elytron based set up.

15.6 Client Authentication with Elytron Client

WildFly Elytron uses the Elytron Client project to enable remote clients to authenticate using Elytron. Elytron Client has the following components:

Component	Description
Authentication Configuration	Contains authentication information such as usernames, passwords, allowed SASL mechanisms, and the security realm to use during digest authentication.
MatchRule	Rule used for deciding which authentication configuration to use.
Authentication Context	Set of rules and authentication configurations to use with a client for establishing a connection.

When a connection is established, the client makes use of an authentication context, which gives rules that match which authentication configuration is used with an outbound connection. For example, you could have a rules that use one authentication configuration when connecting to *server1* and another authentication configuration when connecting with *server2*. The authentication context is comprised of a set of authentication configurations and a set of rules that define how they are selected when establishing a connection. An authentication context can also reference *ssl-context* and can be matched with rules.

To create a client that uses security information when establishing a connection:

- Create one or more authentication configurations.
- Create an authentication context by creating rule and authentication configuration pairs.
- Create a runnable for establishing your connection.
- Use your authentication context to run your runnable.

When you establish your connection, Elytron Client will use the set of rules provided by the authentication context to match the correct authentication configuration to use during authentication.

You can use one of the following approaches to use security information when establishing a client connection.

IMPORTANT: When using Elytron Client to make EJB calls, any hard-coded programatic authentication information, such as setting *Context.SECURITY_PRINCIPAL* in the *javax.naming.InitialContext*, will override the Elytron Client configuration.

15.6.1 The Configuration File Approach

The configuration file approach involves creating an XML file with your authentication configuration, authentication context, and match rules.

custom-config.xml

```
<configuration>
    <authentication-client xmlns="urn:elytron:1.0">
        <authentication-rules>
            <rule use-configuration="monitor">
                <match-host name="127.0.0.1" />
            </rule>
            <rule use-configuration="administrator">
                <match-host name="localhost" />
            </rule>
        </authentication-rules>
        <authentication-configurations>
            <configuration name="monitor">
                 <sasl-mechanism-selector selector="DIGEST-MD5"/>
                 <providers>
                     <use-service-loader />
                 </providers>
                 <set-user-name name="monitor" />
                 <credentials>
                     <clear-password password="password1!" />
                 </credentials>
                 <set-mechanism-realm name="ManagementRealm" />
             </configuration>

             <configuration name="administrator">
                 <sasl-mechanism-selector selector="DIGEST-MD5"/>
                 <providers>
                     <use-service-loader />
                 </providers>
                 <set-user-name name="administrator" />
                 <credentials>
                     <clear-password password="password1!" />
                 </credentials>
                 <set-mechanism-realm name="ManagementRealm" />
             </configuration>
        </authentication-configurations>
    </authentication-client>
</configuration>
```

You can then reference that file in your client's code by setting a system property when running your client.

```
$ java -Dwildfly.config.url=/path/to/the.xml …..
```

IMPORTANT: If you use the The Programmatic Approach, it will override any provided configuration files even if the *wildfly.config.url* system property is set.

When creating rules, you can look for matches on various parameters such as hostname, port, protocol, or username. A full list of options for *MatchRule* are available in the Javadocs. Rules are evaluated in the order in which they are configured.

Common Rules

Rule	Description
match-domain	Takes a single *name* attribute specifying the security domain to match against.
match-host	Takes a single *name* attribute specifying the hostname to match against. For example, the host *127.0.0.1* would match on http://127.0.0.1:9990/my/path.
match-no-userinfo	Matches against URIs with no userinfo.
match-path	Takes a single *name* attribute specifying the path to match against. For example, the path */my/path/* would match on http://127.0.0.1:9990/my/path.
match-port	Takes a single *name* attribute specifying the port to match against. For example, the port *9990* would match on http://127.0.0.1:9990/my/path.
match-protocol	Takes a single *name* attribute specifying the protocol to match against. For example, the protocol *http* would match on http://127.0.0.1:9990/my/path.
match-purpose	Takes a *names* attribute specifying the list of purposes to match against.
match-urn	Takes a single *name* attribute specifying the URN to match against.
match-userinfo	Takes a single *name* attribute specifying the userinfo to match against.

15.6.2 The Programmatic Approach

The programatic approach configures all the Elytron Client configuration in the client's code:

```
//create your authentication configuration
AuthenticationConfiguration adminConfig =
    AuthenticationConfiguration.empty()
      .setSaslMechanismSelector(SaslMechanismSelector.NONE.addMechanism("DIGEST-MD5"))
      .useRealm("ManagementRealm")
      .useName("administrator")
      .usePassword("password1!");

//create your authentication context
AuthenticationContext context = AuthenticationContext.empty();
context = context.with(MatchRule.ALL.matchHost("127.0.0.1"), adminConfig);


//create your runnable for establishing a connection
Runnable runnable =
    new Runnable() {
      public void run() {
        try {
           //Establish your connection and do some work
        } catch (Exception e) {
          e.printStackTrace();
        }
      }
    };

//use your authentication context to run your client
context.run(runnable);
```

When adding configuration details to *AuthenticationConfiguration* and *AuthenticationContext*, each method call returns a new instance of that object. For example, if you wanted separate configurations when connecting over different hostnames, you could do the following:

```
//create your authentication configuration
AuthenticationConfiguration commonConfig =
    AuthenticationConfiguration.empty()
      .setSaslMechanismSelector(SaslMechanismSelector.NONE.addMechanism("DIGEST-MD5"))
      .useRealm("ManagementRealm");

AuthenticationConfiguration administrator =
    commonConfig
      .useName("administrator")
      .usePassword("password1!");


AuthenticationConfiguration monitor =
    commonConfig
      .useName("monitor")
      .usePassword("password1!");


//create your authentication context
AuthenticationContext context = AuthenticationContext.empty();
context = context.with(MatchRule.ALL.matchHost("127.0.0.1"), administrator);
context = context.with(MatchRule.ALL.matchHost("localhost"), monitor);
```

Common Rules

Rule	Description
matchLocalSecurityDomain(String name)	This is the same as match-domain in the configuration file approach.
matchNoUser()	This is the same as match-no-user in the configuration file approach.
matchPath(String pathSpec)	This is the same as match-path in the configuration file approach.
matchPort(int port)	This is the same as match-port in the configuration file approach.
matchProtocol(String protoName)	This is the same as match-port in the configuration file approach.
matchPurpose(String purpose)	Create a new rule which is the same as this rule, but also matches the given purpose name.
matchPurposes(String... purposes)	This is the same as match-purpose in the configuration file approach.
matchUrnName(String name)	This is the same as match-urn in the configuration file approach.
matchUser(String userSpec)	This is the same as match-userinfo in the configuration file approach.

Also, instead of starting with an empty authentication configuration, you can start with the current configured one by using *captureCurrent()*.

```
//create your authentication configuration
AuthenticationConfiguration commonConfig = AuthenticationConfiguration.captureCurrent();
```

Using *captureCurrent()* will capture any previously established authentication context and use it as your new base configuration. A authentication context is established once its been activated by calling *run()*. If *captureCurrent()* is called and no context is currently active, it will try and use the default authentication if available. You can find more details about this in The Configuration File Approach, The Default Configuration Approach, and Using Elytron Client with Clients Deployed to WildFly sections.

Using *AuthenticationConfiguration.EMPTY* should only be used as a base to build a configuration on top of and should not be used on its own. It provides a configuration that uses the JVM-wide registered providers and enables anonymous authentication.

When specifying the providers on top of the *AuthenticationConfiguration.EMPTY* configuration, you can specify a custom list, but most users should use *WildFlyElytronProvider()* providers.

When creating an authentication context, using the *context.with(...)* will create a new context that merges the rules and authentication configuration from the current context with the provided rule and authentication configuration. The provided rule and authentication configuration will appear after the ones in the current context.

15.6.3 The Default Configuration Approach

The default configuration approach relies completely on the configuration provided by Elytron Client:

```
//create your runnable for establishing a connection
Runnable runnable =
    new Runnable() {
      public void run() {
        try {
           //Establish your connection and do some work
        } catch (Exception e) {
          e.printStackTrace();
        }
      }
    };

// run runnable directly
runnable.run();
```

To provide a default configuration, Elytron Client tries to auto-discover a *wildfly-config.xml* file on the filesystem. It looks in the following locations:

- Location specified by the *wildfly.config.url* system property set outside of the client code.
- The classpath root directory.
- The *META-INF* directory on the classpath.

If it does not find one, it will try and use the default *wildfly-config.xml* provided in the *$WILDFLY_HOME/bin/client/jboss-client.jar*.

default wildfly-config.xml

```
<configuration>
  <authentication-client xmlns="urn:elytron:1.0">
    <authentication-rules>
      <rule use-configuration="default" />
    </authentication-rules>
    <authentication-configurations>
      <configuration name="default">
        <sasl-mechanism-selector selector="#ALL" />
        <set-mechanism-properties>
          <property key="wildfly.sasl.local-user.quiet-auth" value="true" />
        </set-mechanism-properties>
        <providers>
            <use-service-loader />
        </providers>
      </configuration>
    </authentication-configurations>
  </authentication-client>
</configuration>
```

15.6.4 Using Elytron Client with Clients Deployed to WildFly

Clients deployed to WildFly can also make use of Elytron Client. In cases where you have included a *wildfly-config.xml* with your deployment or the system property has been set, an *AuthenticationContext* is automatically parsed and created from that file.

To load a configuration file outside of the deployment, you can use the *parseAuthenticationClientConfiguration(URI)* method. This method will return an *AuthentcationContext* which you can then use in your client's code using the The Programmatic Approach.

Additionally, clients will also automatically parse and create an AuthenticationContext from the client configuration provided by the *elytron* subsystem. The client configuration in the *elytron* subsystem can also take advantage of other components defined in the *elytron* subsystem such as credential stores. If client configuration is provided by BOTH the deployment and the *elytron* subsystem, the *elytron* subsystem's configuration is used.

15.6.5 Client configuration using wildfly-config.xml

Prior to WildFly 11, many WildFly client libraries used different configuration strategies. WildFly 11 introduces a new `wildfly-config.xml` file which unifies all client configuration in a single place. In addition to being able to configure authentication using Elytron as described in the previous section, a `wildfly-config.xml` file can also be used to:

Configure EJB client connections, global interceptors, and invocation timeout

Schema location:
https://github.com/wildfly/jboss-ejb-client/blob/4.0.2.Final/src/main/resources/schema/wildfly-client-ejb_3_0.xs

Example configuration:

wildfly-config.xml

```
<configuration>
...
    <jboss-ejb-client xmlns="urn:jboss:wildfly-client-ejb:3.0">
        <invocation-timeout seconds="10"/>
        <connections>
            <connection uri="remote+http://10.20.30.40:8080"/>
        </connections>
        <global-interceptors>
            <interceptor class="org.jboss.example.ExampleInterceptor"/>
        </global-interceptors>
    </jboss-ejb-client>
...
</configuration>
```

Configure HTTP client

Schema location:
https://github.com/wildfly/wildfly-http-client/blob/1.0.2.Final/common/src/main/resources/schema/wildfly-http-c

Example configuration:

wildfly-config.xml

```
<configuration>
...
    <http-client xmlns="urn:wildfly-http-client:1.0">
        <defaults>
            <eagerly-acquire-session value="true" />
            <buffer-pool buffer-size="2000" max-size="10" direct="true" thread-local-size="1" />
        </defaults>
    </http-client>
...
</configuration>
```

Configure a remoting endpoint

Schema location:
https://github.com/jboss-remoting/jboss-remoting/blob/5.0.1.Final/src/main/resources/schema/jboss-remoting_

Example configuration:

wildfly-config.xml

```
<configuration>
...
    <endpoint xmlns="urn:jboss-remoting:5.0">
        <connections>
            <connection destination="remote+http://10.20.30.40:8080" read-timeout="50"
write-timeout="50" heartbeat-interval="10000"/>
        </connections>
    </endpoint>
...
</configuration>
```

Configure the default XNIO worker

Schema location:https://github.com/xnio/xnio/blob/3.5.1.Final/api/src/main/resources/schema/xnio_3_5.xsd

Example configuration:

wildfly-config.xml

```
<configuration>
...
    <worker xmlns="urn:xnio:3.5">
        <io-threads value="10"/>
        <task-keepalive value="100"/>
        <stack-size value="5000"/>
    </worker>
...
</configuration>
```


Note that WildFly client libraries do have reasonable default configuration. Thus, adding configuration for these clients to `wildfly-config.xml` isn't mandatory.

Couldn't find a page to include called: KeyCloak Integration

15.7 Client Authentication with Elytron Client

15.7.1 Client Authentication with Elytron Client

WildFly Elytron uses the Elytron Client project to enable remote clients to authenticate using Elytron. Elytron Client has the following components:

Component	Description
Authentication Configuration	Contains authentication information such as usernames, passwords, allowed SASL mechanisms, and the security realm to use during digest authentication.
MatchRule	Rule used for deciding which authentication configuration to use.
Authentication Context	Set of rules and authentication configurations to use with a client for establishing a connection.

When a connection is established, the client makes use of an authentication context, which gives rules that match which authentication configuration is used with an outbound connection. For example, you could have a rules that use one authentication configuration when connecting to *server1* and another authentication configuration when connecting with *server2*. The authentication context is comprised of a set of authentication configurations and a set of rules that define how they are selected when establishing a connection. An authentication context can also reference *ssl-context* and can be matched with rules.

To create a client that uses security information when establishing a connection:

- Create one or more authentication configurations.
- Create an authentication context by creating rule and authentication configuration pairs.
- Create a runnable for establishing your connection.
- Use your authentication context to run your runnable.

When you establish your connection, Elytron Client will use the set of rules provided by the authentication context to match the correct authentication configuration to use during authentication.

You can use one of the following approaches to use security information when establishing a client connection.

IMPORTANT: When using Elytron Client to make EJB calls, any hard-coded programatic authentication information, such as setting *Context.SECURITY_PRINCIPAL* in the *javax.naming.InitialContext*, will override the Elytron Client configuration.

The Configuration File Approach

The configuration file approach involves creating an XML file with your authentication configuration, authentication context, and match rules.

custom-config.xml

```
<configuration>
    <authentication-client xmlns="urn:elytron:1.0">
        <authentication-rules>
            <rule use-configuration="monitor">
                <match-host name="127.0.0.1" />
            </rule>
            <rule use-configuration="administrator">
                <match-host name="localhost" />
            </rule>
        </authentication-rules>
        <authentication-configurations>
            <configuration name="monitor">
                 <sasl-mechanism-selector selector="DIGEST-MD5"/>
                 <providers>
                     <use-service-loader />
                 </providers>
                 <set-user-name name="monitor" />
                 <credentials>
                     <clear-password password="password1!" />
                 </credentials>
                 <set-mechanism-realm name="ManagementRealm" />
             </configuration>

             <configuration name="administrator">
                 <sasl-mechanism-selector selector="DIGEST-MD5"/>
                 <providers>
                     <use-service-loader />
                 </providers>
                 <set-user-name name="administrator" />
                 <credentials>
                     <clear-password password="password1!" />
                 </credentials>
                 <set-mechanism-realm name="ManagementRealm" />
             </configuration>
        </authentication-configurations>
    </authentication-client>
</configuration>
```

You can then reference that file in your client's code by setting a system property when running your client.

```
$ java -Dwildfly.config.url=/path/to/the.xml …..
```

IMPORTANT: If you use the The Programmatic Approach, it will override any provided configuration files even if the *wildfly.config.url* system property is set.

When creating rules, you can look for matches on various parameters such as hostname, port, protocol, or username. A full list of options for *MatchRule* are available in the Javadocs. Rules are evaluated in the order in which they are configured.

Common Rules

Rule	Description
match-domain	Takes a single *name* attribute specifying the security domain to match against.
match-host	Takes a single *name* attribute specifying the hostname to match against. For example, the host *127.0.0.1* would match on http://127.0.0.1:9990/my/path.
match-no-userinfo	Matches against URIs with no userinfo.
match-path	Takes a single *name* attribute specifying the path to match against. For example, the path */my/path/* would match on http://127.0.0.1:9990/my/path.
match-port	Takes a single *name* attribute specifying the port to match against. For example, the port *9990* would match on http://127.0.0.1:9990/my/path.
match-protocol	Takes a single *name* attribute specifying the protocol to match against. For example, the protocol *http* would match on http://127.0.0.1:9990/my/path.
match-purpose	Takes a *names* attribute specifying the list of purposes to match against.
match-urn	Takes a single *name* attribute specifying the URN to match against.
match-userinfo	Takes a single *name* attribute specifying the userinfo to match against.

The Programmatic Approach

The programatic approach configures all the Elytron Client configuration in the client's code:

```
//create your authentication configuration
AuthenticationConfiguration adminConfig =
    AuthenticationConfiguration.empty()
      .setSaslMechanismSelector(SaslMechanismSelector.NONE.addMechanism("DIGEST-MD5"))
      .useRealm("ManagementRealm")
      .useName("administrator")
      .usePassword("password1!");

//create your authentication context
AuthenticationContext context = AuthenticationContext.empty();
context = context.with(MatchRule.ALL.matchHost("127.0.0.1"), adminConfig);


//create your runnable for establishing a connection
Runnable runnable =
    new Runnable() {
      public void run() {
        try {
           //Establish your connection and do some work
        } catch (Exception e) {
          e.printStackTrace();
        }
      }
    };

//use your authentication context to run your client
context.run(runnable);
```

When adding configuration details to *AuthenticationConfiguration* and *AuthenticationContext*, each method call returns a new instance of that object. For example, if you wanted separate configurations when connecting over different hostnames, you could do the following:

```
//create your authentication configuration
AuthenticationConfiguration commonConfig =
    AuthenticationConfiguration.empty()
      .setSaslMechanismSelector(SaslMechanismSelector.NONE.addMechanism("DIGEST-MD5"))
      .useRealm("ManagementRealm");

AuthenticationConfiguration administrator =
    commonConfig
      .useName("administrator")
      .usePassword("password1!");


AuthenticationConfiguration monitor =
    commonConfig
      .useName("monitor")
      .usePassword("password1!");


//create your authentication context
AuthenticationContext context = AuthenticationContext.empty();
context = context.with(MatchRule.ALL.matchHost("127.0.0.1"), administrator);
context = context.with(MatchRule.ALL.matchHost("localhost"), monitor);
```

Common Rules

Rule	Description
matchLocalSecurityDomain(String name)	This is the same as match-domain in the configuration file approach.
matchNoUser()	This is the same as match-no-user in the configuration file approach.
matchPath(String pathSpec)	This is the same as match-path in the configuration file approach.
matchPort(int port)	This is the same as match-port in the configuration file approach.
matchProtocol(String protoName)	This is the same as match-port in the configuration file approach.
matchPurpose(String purpose)	Create a new rule which is the same as this rule, but also matches the given purpose name.
matchPurposes(String... purposes)	This is the same as match-purpose in the configuration file approach.
matchUrnName(String name)	This is the same as match-urn in the configuration file approach.
matchUser(String userSpec)	This is the same as match-userinfo in the configuration file approach.

Also, instead of starting with an empty authentication configuration, you can start with the current configured one by using *captureCurrent()*.

```
//create your authentication configuration
AuthenticationConfiguration commonConfig = AuthenticationConfiguration.captureCurrent();
```

Using *captureCurrent()* will capture any previously established authentication context and use it as your new base configuration. A authentication context is established once its been activated by calling *run()*. If *captureCurrent()* is called and no context is currently active, it will try and use the default authentication if available. You can find more details about this in The Configuration File Approach, The Default Configuration Approach, and Using Elytron Client with Clients Deployed to WildFly sections.

Using *AuthenticationConfiguration.EMPTY* should only be used as a base to build a configuration on top of and should not be used on its own. It provides a configuration that uses the JVM-wide registered providers and enables anonymous authentication.

When specifying the providers on top of the *AuthenticationConfiguration.EMPTY* configuration, you can specify a custom list, but most users should use *WildFlyElytronProvider()* providers.

When creating an authentication context, using the *context.with(...)* will create a new context that merges the rules and authentication configuration from the current context with the provided rule and authentication configuration. The provided rule and authentication configuration will appear after the ones in the current context.

The Default Configuration Approach

The default configuration approach relies completely on the configuration provided by Elytron Client:

```
//create your runnable for establishing a connection
Runnable runnable =
    new Runnable() {
      public void run() {
        try {
           //Establish your connection and do some work
        } catch (Exception e) {
          e.printStackTrace();
        }
      }
    };

// run runnable directly
runnable.run();
```

To provide a default configuration, Elytron Client tries to auto-discover a *wildfly-config.xml* file on the filesystem. It looks in the following locations:

- Location specified by the *wildfly.config.url* system property set outside of the client code.
- The classpath root directory.
- The *META-INF* directory on the classpath.

If it does not find one, it will try and use the default *wildfly-config.xml* provided in the *$WILDFLY_HOME/bin/client/jboss-client.jar*.

default wildfly-config.xml

```
<configuration>
  <authentication-client xmlns="urn:elytron:1.0">
    <authentication-rules>
      <rule use-configuration="default" />
    </authentication-rules>
    <authentication-configurations>
      <configuration name="default">
        <sasl-mechanism-selector selector="#ALL" />
        <set-mechanism-properties>
          <property key="wildfly.sasl.local-user.quiet-auth" value="true" />
        </set-mechanism-properties>
        <providers>
            <use-service-loader />
        </providers>
      </configuration>
    </authentication-configurations>
  </authentication-client>
</configuration>
```

Using Elytron Client with Clients Deployed to WildFly

Clients deployed to WildFly can also make use of Elytron Client. In cases where you have included a *wildfly-config.xml* with your deployment or the system property has been set, an *AuthenticationContext* is automatically parsed and created from that file.

To load a configuration file outside of the deployment, you can use the *parseAuthenticationClientConfiguration(URI)* method. This method will return an *AuthentcationContext* which you can then use in your client's code using the The Programmatic Approach.

Additionally, clients will also automatically parse and create an AuthenticationContext from the client configuration provided by the *elytron* subsystem. The client configuration in the *elytron* subsystem can also take advantage of other components defined in the *elytron* subsystem such as credential stores. If client configuration is provided by BOTH the deployment and the *elytron* subsystem, the *elytron* subsystem's configuration is used.

Client configuration using wildfly-config.xml

Prior to WildFly 11, many WildFly client libraries used different configuration strategies. WildFly 11 introduces a new `wildfly-config.xml` file which unifies all client configuration in a single place. In addition to being able to configure authentication using Elytron as described in the previous section, a `wildfly-config.xml` file can also be used to:

Configure EJB client connections, global interceptors, and invocation timeout

Schema location:
https://github.com/wildfly/jboss-ejb-client/blob/4.0.2.Final/src/main/resources/schema/wildfly-client-ejb_3_0.xs

Example configuration:

wildfly-config.xml

```
<configuration>
...
    <jboss-ejb-client xmlns="urn:jboss:wildfly-client-ejb:3.0">
        <invocation-timeout seconds="10"/>
        <connections>
            <connection uri="remote+http://10.20.30.40:8080"/>
        </connections>
        <global-interceptors>
            <interceptor class="org.jboss.example.ExampleInterceptor"/>
        </global-interceptors>
    </jboss-ejb-client>
...
</configuration>
```

Configure HTTP client

Schema location:
https://github.com/wildfly/wildfly-http-client/blob/1.0.2.Final/common/src/main/resources/schema/wildfly-http-c

Example configuration:

wildfly-config.xml

```
<configuration>
...
    <http-client xmlns="urn:wildfly-http-client:1.0">
        <defaults>
            <eagerly-acquire-session value="true" />
            <buffer-pool buffer-size="2000" max-size="10" direct="true" thread-local-size="1" />
        </defaults>
    </http-client>
...
</configuration>
```

Configure a remoting endpoint

Schema location:
https://github.com/jboss-remoting/jboss-remoting/blob/5.0.1.Final/src/main/resources/schema/jboss-remoting_

Example configuration:

wildfly-config.xml

```
<configuration>
...
    <endpoint xmlns="urn:jboss-remoting:5.0">
        <connections>
            <connection destination="remote+http://10.20.30.40:8080" read-timeout="50"
write-timeout="50" heartbeat-interval="10000"/>
        </connections>
    </endpoint>
...
</configuration>
```

Configure the default XNIO worker

Schema location:https://github.com/xnio/xnio/blob/3.5.1.Final/api/src/main/resources/schema/xnio_3_5.xsd

Example configuration:

wildfly-config.xml

```
<configuration>
...
    <worker xmlns="urn:xnio:3.5">
        <io-threads value="10"/>
        <task-keepalive value="100"/>
        <stack-size value="5000"/>
    </worker>
...
</configuration>
```

Note that WildFly client libraries do have reasonable default configuration. Thus, adding configuration for these clients to `wildfly-config.xml` isn't mandatory.

15.7.2 Client configuration using wildfly-config.xml

Prior to WildFly 11, many WildFly client libraries used different configuration strategies. WildFly 11 introduces a new `wildfly-config.xml` file which unifies all client configuration in a single place. In addition to being able to configure authentication using Elytron as described in the previous section, a `wildfly-config.xml` file can also be used to:

Configure EJB client connections, global interceptors, and invocation timeout

Schema location:
https://github.com/wildfly/jboss-ejb-client/blob/4.0.2.Final/src/main/resources/schema/wildfly-client-ejb_3_0.xs

Example configuration:

wildfly-config.xml

```
<configuration>
...
    <jboss-ejb-client xmlns="urn:jboss:wildfly-client-ejb:3.0">
        <invocation-timeout seconds="10"/>
        <connections>
            <connection uri="remote+http://10.20.30.40:8080"/>
        </connections>
        <global-interceptors>
            <interceptor class="org.jboss.example.ExampleInterceptor"/>
        </global-interceptors>
    </jboss-ejb-client>
...
</configuration>
```

Configure HTTP client

Schema location:
https://github.com/wildfly/wildfly-http-client/blob/1.0.2.Final/common/src/main/resources/schema/wildfly-http-c

Example configuration:

wildfly-config.xml

```
<configuration>
...
    <http-client xmlns="urn:wildfly-http-client:1.0">
        <defaults>
            <eagerly-acquire-session value="true" />
            <buffer-pool buffer-size="2000" max-size="10" direct="true" thread-local-size="1" />
        </defaults>
    </http-client>
...
</configuration>
```

Configure a remoting endpoint

Schema location:
https://github.com/jboss-remoting/jboss-remoting/blob/5.0.1.Final/src/main/resources/schema/jboss-remoting_

Example configuration:

wildfly-config.xml

```
<configuration>
...
    <endpoint xmlns="urn:jboss-remoting:5.0">
        <connections>
            <connection destination="remote+http://10.20.30.40:8080" read-timeout="50"
write-timeout="50" heartbeat-interval="10000"/>
        </connections>
    </endpoint>
...
</configuration>
```

Configure the default XNIO worker

Schema location:https://github.com/xnio/xnio/blob/3.5.1.Final/api/src/main/resources/schema/xnio_3_5.xsd

Example configuration:

wildfly-config.xml

```
<configuration>
...
    <worker xmlns="urn:xnio:3.5">
        <io-threads value="10"/>
        <task-keepalive value="100"/>
        <stack-size value="5000"/>
    </worker>
...
</configuration>
```

Note that WildFly client libraries do have reasonable default configuration. Thus, adding configuration for these clients to `wildfly-config.xml` isn't mandatory.

15.8 Elytron and Java Authorization Contract for Containers (JACC)

15.8.1 Overview

This document will guide you on how to enable JACC using Elytron Subsystem.

15.8.2 Disabling JACC in Legacy Security Subsystem (PicketBox)

By default, the application server uses the legacy security subsystem (PicketBox) to configure the JACC policy provider and factory. The default configuration maps to implementations from PicketBox.

In order to use Elytron to manage JACC configuration (or any other policy you want to install to the application server) you must first disable JACC in legacy security subsystem. For that, please execute the following CLI command:

```
/subsystem=security:write-attribute(name=initialize-jacc, value=false)
```

The command above tells the legacy security subsystem to not initialize any JACC related configuration, but rely on the policies defined via Elytron subsystem as we'll see in the next sections.

15.8.3 Defining a JACC Policy Provider

Elytron subsystem provides a built-in policy provider based on JACC specification. To create the policy provider you can execute a CLI command as follows:

```
[standalone@localhost:9990 /] /subsystem=elytron/policy=jacc:add(jacc-policy={})
```

After executing the command above, please reload the server configuration as follows:

```
[standalone@localhost:9990 /] reload
```

15.8.4 Enabling JACC to a Web Deployment

Once JACC Policy Provider is defined you can enable JACC to web deployments by executing the following command:

```
[standalone@localhost:9990 /]
/subsystem=undertow/application-security-domain=other:add(http-authentication-factory=application-
```

The command above defines a default security domain for applications if none is provided in **jboss-web.xml.** In case you already have a **application-security-domain** defined and just want to enable JACC you can execute a command as follows:

```
[standalone@localhost:9990 /]
/subsystem=undertow/application-security-domain=my-security-domain:write-attribute(name=enable-jac
```

15.8.5 Enabling JACC to a EJB Deployment

Once JACC Policy Provider is defined you can enable JACC to EJB deployments by executing the following command:

```
[standalone@localhost:9990 /]
/subsystem=ejb3/application-security-domain=other:add(security-domain=ApplicationDomain,enable-jac
```

The command above defines a default security domain for EJBs. In case you already have a *application-security-domain *defined and just want to enable JACC you can execute a command as follows:

```
[standalone@localhost:9990 /]
/subsystem=ejb3/application-security-domain=my-security-domain:write-attribute(name=enable-jacc,va
```

15.9 Elytron Subsystem

WildFly Elytron is a security framework used to unify security across the entire application server. The *elytron* subsystem enables a single point of configuration for securing both applications and the management interfaces. WildFly Elytron also provides a set of APIs and SPIs for providing custom implementations of functionality and integrating with the *elytron* subsystem.

In addition, there are several other important features of the WildFly Elytron:

- Stronger authentication mechanisms for HTTP and SASL authentication.
- Improved architecture that allows for *SecurityIdentities* to be propagated across security domains and transparently transformed ready to be used for authorization. This transformation takes place using configurable role decoders, role mappers, and permission mappers.
- Centralized point for SSL/TLS configuration including cipher suites and protocols.
- SSL/TLS optimizations such as eager *SecureIdentity* construction and closely tying authorization to establishing an SSL/TLS connection. Eager *SecureIdentity* construction eliminates the need for a *SecureIdentity* to be constructed on a per-request basis. Closely tying authentication to establishing an SSL/TLS connection enables permission checks to happen *BEFORE* the first request is received.
- A secure credential store that replaces the previous vault implementation to store clear text credentials.

The new *elytron* subsystem exists in parallel to the legacy *security* subsystem and legacy core management authentication. Both the legacy and Elytron methods may be used for securing the management interfaces as well as providing security for applications.

15.9.1 Get Started using the Elytron Subsystem

To get started using Elytron, refer to these topics:

- Use the default Elytron components for application and management authentication
- Secure an application with a new identity store stored in a filesystem or database.
- Set up one-way SSL/TLS for applications or the management interfaces.
- Set up two-way SSL/TLS for applications or the management interfaces.
- Create a credential store and use it with your SSL/TLS configuration.
- Use certificate-based authentication with applications.
- Override an application's authentication configuration with Elytron authentication.
- Configure Kerberos authentication for applications.
- Secure applications and the management interfaces with an LDAP-based identity store.

15.9.2 Provided components

Wildfly Elytron provides a default set of implementations in the *elytron* subsystem.

Factories

Component	Description
aggregate-http-server-mechanism-factory	An HTTP server factory definition where the HTTP server factory is an aggregation of other HTTP server factories.
aggregate-sasl-server-factory	A SASL server factory definition where the SASL server factory is an aggregation of other SASL server factories.
configurable-http-server-mechanism-factory	A SASL server factory definition where the SASL server factory is an aggregation of other SASL server factories.
configurable-sasl-server-factory	A SASL server factory definition where the SASL server factory is an aggregation of other SASL server factories.
custom-credential-security-factory	A custom credential *SecurityFactory* definition.
http-authentication-factory	Resource containing the association of a security domain with a *HttpServerAuthenticationMechanismFactory*.
kerberos-security-factory	A security factory for obtaining a *GSSCredential* for use during authentication.
mechanism-provider-filtering-sasl-server-factory	A SASL server factory definition that enables filtering by provider where the factory was loaded using a provider.
provider-http-server-mechanism-factory	An HTTP server factory definition where the HTTP server factory is an aggregation of factories from the provider list.
provider-sasl-server-factory	A SASL server factory definition where the SASL server factory is an aggregation of factories from the provider list.
sasl-authentication-factory	Resource containing the association of a security domain with a *SaslServerFactory*.
service-loader-http-server-mechanism-factory	An HTTP server factory definition where the HTTP server factory is an aggregation of factories identified using a *ServiceLoader*
service-loader-sasl-server-factory	A SASL server factory definition where the SASL server factory is an aggregation of factories identified using a *ServiceLoader*

Principal Transformers

Component	Description
aggregate-principal-transformer	A principal transformer definition where the principal transformer is an aggregation of other principal transformers.
chained-principal-transformer	A principal transformer definition where the principal transformer is a chaining of other principal transformers.
constant-principal-transformer	A principal transformer definition where the principal transformer always returns the same constant.
custom-principal-transformer	A custom principal transformer definition.
regex-principal-transformer	A regular expression based principal transformer
regex-validating-principal-transformer	A regular expression based principal transformer which uses the regular expression to validate the name.

Principal Decoders

Component	Description
aggregate-principal-decoder	A principal decoder definition where the principal decoder is an aggregation of other principal decoders.
concatenating-principal-decoder	A principal decoder definition where the principal decoder is a concatenation of other principal decoders.
constant-principal-decoder	Definition of a principal decoder that always returns the same constant.
custom-principal-decoder	Definition of a custom principal decoder.
x500-attribute-principal-decoder	Definition of a X500 attribute based principal decoder.

Realm Mappers

Component	Description
constant-realm-mapper	Definition of a constant realm mapper that always returns the same value.
custom-realm-mapper	Definition of a custom realm mapper
mapped-regex-realm-mapper	Definition of a realm mapper implementation that first uses a regular expression to extract the realm name, this is then converted using the configured mapping of realm names.
simple-regex-realm-mapper	Definition of a simple realm mapper that attempts to extract the realm name using the capture group from the regular expression, if that does not provide a match then the delegate realm mapper is used instead.

Realms

Component	Description
aggregate-realm	A realm definition that is an aggregation of two realms, one for the authentication steps and one for loading the identity for the authorization steps.
caching-realm	A realm definition that enables caching to another security realm. Caching strategy is *Least Recently Used* where least accessed entries are discarded when maximum number of entries is reached.
custom-modifiable-realm	Custom realm configured as being modifiable will be expected to implement the *ModifiableSecurityRealm* interface. By configuring a realm as being modifiable management operations will be made available to manipulate the realm.
custom-realm	A custom realm definitions can implement either the s *SecurityRealm* interface or the *ModifiableSecurityRealm* interface. Regardless of which interface is implemented management operations will not be exposed to manage the realm. However other services that depend on the realm will still be able to perform a type check and cast to gain access to the modification API.
filesystem-realm	A simple security realm definition backed by the filesystem.
identity-realm	A security realm definition where identities are represented in the management model.
jdbc-realm	A security realm definition backed by database using JDBC.
key-store-realm	A security realm definition backed by a keystore.
ldap-realm	A security realm definition backed by LDAP.
properties-realm	A security realm definition backed by properties files.
token-realm	A security realm definition capable of validating and extracting identities from security tokens.
trust-managers	A trust manager definition for creating the *TrustManager* list as used to create an SSL context.

Permission Mappers

Component	Description
custom-permission-mapper	Definition of a custom permission mapper.
logical-permission-mapper	Definition of a logical permission mapper.
simple-permission-mapper	Definition of a simple configured permission mapper.
constant-permission-mapper	Definition of a permission mapper that always returns the same constant.

Role Decoders

Component	Description
custom-role-decoder	Definition of a custom RoleDecoder
simple-role-decoder	Definition of a simple RoleDecoder that takes a single attribute and maps it directly to roles.

Role Mappers

Component	Description
add-prefix-role-mapper	A role mapper definition for a role mapper that adds a prefix to each provided.
add-suffix-role-mapper	A role mapper definition for a role mapper that adds a suffix to each provided.
constant-role-mapper	A role mapper definition where a constant set of roles is always returned.
aggregate-role-mapper	A role mapper definition where the role mapper is an aggregation of other role mappers.
logical-role-mapper	A role mapper definition for a role mapper that performs a logical operation using two referenced role mappers.
custom-role-mapper	Definition of a custom role mapper

SSL Components

Component	Description
client-ssl-context	An SSLContext for use on the client side of a connection.
filtering-key-store	A filtering keystore definition, which provides a keystore by filtering a *key-store*.
key-managers	A key manager definition for creating the key manager list as used to create an SSL context.
key-store	A keystore definition.
ldap-key-store	An LDAP keystore definition, which loads a keystore from an LDAP server.
server-ssl-context	An SSL context for use on the server side of a connection.

Other

Component	Description
aggregate-providers	An aggregation of two or more *Provider[]* resources.
authentication-configuration	An individual authentication configuration definition, which is used by clients deployed to Wildfly and other resources for authenticating when making a remote connection.
authentication-context	An individual authentication context definition, which is used to supply an *ssl-context* and *authentication-configuration* when clients deployed to Wildfly and other resources make a remoting connection.
credential-store	Credential store to keep alias for sensitive information such as passwords for external services.
dir-context	The configuration to connect to a directory (LDAP) server.
provider-loader	A definition for a provider loader.
security-domain	A security domain definition.
security-property	A definition of a security property to be set.

15.9.3 Out of the Box Configuration

WildFly provides a set of components configured by default. While these components are ready to use, the legacy *security* subsystem and legacy core management authentication is still used by default. To configure WildFly to use the these configured components as well as create new ones, see the Using the Elytron Subsystem section.

Default Component	Description
ApplicationDomain	The *ApplicationDomain* security domain uses *ApplicationRealm* and *groups-to-roles* for authentication. I also uses *default-permission-mapper* to assign the login permission.
ManagementDomain	The *ManagementDomain* security domain uses two secur realms for authentication: *ManagementRealm* with *groups-to-roles* and *local* with *super-user-mapper*. It also u *default-permission-mapper* to assign the login permission.
local (security realm)	The *local* security realm does no authentication and sets t identity of principals to *$local*

ApplicationRealm	The *ApplicationRealm* security realm is a properties realm authenticates principals using *application-users.properties* assigns roles using *application-roles.properties*. These file are located under *jboss.server.config.dir*, which by default maps to *EAP_HOME/standalone/configuration*. They are a the same files used by the legacy security default configuration.
ManagementRealm	The *ManagementRealm* security realm is a properties rea that authenticates principals using *mgmt-users.properties* assigns roles using *mgmt-groups.properties*. These files a located under *jboss.server.config.dir*, which by default, ma to *EAP_HOME/standalone/configuration*. They are also th same files used by the legacy security default configuratio
default-permission-mapper	The *default-permission-mapper* mapper is a constant permission mapper that uses *org.wildfly.security.auth.permission.LoginPermission* to as the login permission and *org.wildfly.extension.batch.jberet.deployment.BatchPermis* to assign permission for batch jobs. The batch permission are *start*, *stop*, *restart*, *abandon*, and *read* which aligns wit *javax.batch.operations.JobOperator*.
local (mapper)	The *local* mapper is a constant role mapper that maps to t *local* security realm. This is used to map authentication to *local* security realm.
groups-to-roles	The *groups-to-roles* mapper is a simple-role-decoder that decode the *groups* information of a principal and use it for *role* information.
super-user-mapper	The *super-user-mapper* mapper is a constant role mapper maps the *SuperUser* role to a principal.
management-http-authentication	The *management-http-authentication* http-authentication-factory can be used for doing authentication over http. It uses the *global* provider-http-server-mechanism-factory to filter authentica mechanism and uses *ManagementDomain* for authenticat principals. It accepts the *DIGEST* authentication mechanis and exposes it as *ManagementRealm* to applications.

application-http-authentication	The *application-http-authentication* http-authentication-fac can be used for doing authentication over http. It uses the *global* provider-http-server-mechanism-factory to filter authentication mechanism and uses *ApplicationDomain* fo authenticating principals. It accepts *BASIC* and *FORM* authentication mechanisms and exposes *BASIC* as *Application Realm* to applications.
global (provider-http-server-mechanism-factory)	This is the HTTP server factory mechanism definition used list the provided authentication mechanisms when creating http authentication factory.
management-sasl-authentication	The *management-sasl-authentication* sasl-authentication-factory can be used for authentication using SASL. It uses the *configured* sasl-server-factory to f authentication mechanisms, which also uses the *global* provider-sasl-server-factory to filter by provider names. *management-sasl-authentication* uses the *ManagementDomain* security domain for authentication of principals. It also maps authentication using *JBOSS-LOCAL-USER* mechanisms using the *local* realm mapper and authentication using *DIGEST-MD5* to *ManagementRealm*.
application-sasl-authentication	The *application-sasl-authentication* sasl-authentication-fac can be used for authentication using SASL. It uses the *configured* sasl-server-factory to filter authentication mechanisms, which also uses the *global* provider-sasl-server-factory to filter by provider names. *application-sasl-authentication* uses the *ApplicationDomai* security domain for authentication of principals.
global (provider-sasl-server-factory)	This is the SASL server factory definition used to create S authentication factories.
elytron (mechanism-provider-filtering-sasl-server-factor)	This is used to filter which *sasl-authentication-factory* is us based on the provider. In this case, *elytron* will match on t *WildFlyElytron* provider name.
configured (configurable-sasl-server-factory)	This is used to filter *sasl-authentication-factory* is used bas on the mechanism name. In this case, *configured* will mat on *JBOSS-LOCAL-USER* and *DIGEST-MD5*. It also sets t *wildfly.sasl.local-user.default-user* to *$local*.
combined-providers	Is an aggregate provider that aggreates the *elytron* and *openssl* provider loaders.
elytron	A provider loader

openssl	A provider loader

Default WildFly Configuration

```
/subsystem=elytron:read-resource(recursive=true)
{
    "outcome" => "success",
    "result" => {
        "default-authentication-context" => undefined,
        "final-providers" => undefined,
        "initial-providers" => "combined-providers",
        "add-prefix-role-mapper" => undefined,
        "add-suffix-role-mapper" => undefined,
        "aggregate-http-server-mechanism-factory" => undefined,
        "aggregate-principal-decoder" => undefined,
        "aggregate-principal-transformer" => undefined,
        "aggregate-providers" => {"combined-providers" => {"providers" => [
            "elytron",
            "openssl"
        ]}},
        "aggregate-realm" => undefined,
        "aggregate-role-mapper" => undefined,
        "aggregate-sasl-server-factory" => undefined,
        "authentication-configuration" => undefined,
        "authentication-context" => undefined,
        "caching-realm" => undefined,
        "chained-principal-transformer" => undefined,
        "client-ssl-context" => undefined,
        "concatenating-principal-decoder" => undefined,
        "configurable-http-server-mechanism-factory" => undefined,
        "configurable-sasl-server-factory" => {"configured" => {
            "filters" => [
                {"pattern-filter" => "JBOSS-LOCAL-USER"},
                {"pattern-filter" => "DIGEST-MD5"}
            ],
            "properties" => {"wildfly.sasl.local-user.default-user" => "$local"},
            "protocol" => undefined,
            "sasl-server-factory" => "elytron",
            "server-name" => undefined
        }},
        "constant-permission-mapper" => {"default-permission-mapper" => {"permissions" => [
            {"class-name" => "org.wildfly.security.auth.permission.LoginPermission"},
            {
                "class-name" => "org.wildfly.extension.batch.jberet.deployment.BatchPermission",
                "module" => "org.wildfly.extension.batch.jberet",
                "target-name" => "*"
            }
        ]}},
        "constant-principal-decoder" => undefined,
        "constant-principal-transformer" => undefined,
        "constant-realm-mapper" => {"local" => {"realm-name" => "local"}},
        "constant-role-mapper" => {"super-user-mapper" => {"roles" => ["SuperUser"]}},
        "credential-store" => undefined,
        "custom-credential-security-factory" => undefined,
        "custom-modifiable-realm" => undefined,
```

```
        "custom-permission-mapper" => undefined,
        "custom-principal-decoder" => undefined,
        "custom-principal-transformer" => undefined,
        "custom-realm" => undefined,
        "custom-realm-mapper" => undefined,
        "custom-role-decoder" => undefined,
        "custom-role-mapper" => undefined,
        "dir-context" => undefined,
        "filesystem-realm" => undefined,
        "filtering-key-store" => undefined,
        "http-authentication-factory" => {
            "management-http-authentication" => {
                "http-server-mechanism-factory" => "global",
                "mechanism-configurations" => [{
                    "mechanism-name" => "DIGEST",
                    "mechanism-realm-configurations" => [{"realm-name" => "ManagementRealm"}]
                }],
                "security-domain" => "ManagementDomain"
            },
            "application-http-authentication" => {
                "http-server-mechanism-factory" => "global",
                "mechanism-configurations" => [
                    {
                        "mechanism-name" => "BASIC",
                        "mechanism-realm-configurations" => [{"realm-name" => "Application
Realm"}]
                    },
                    {"mechanism-name" => "FORM"}
                ],
                "security-domain" => "ApplicationDomain"
            }
        },
        "identity-realm" => {"local" => {
            "attribute-name" => undefined,
            "attribute-values" => undefined,
            "identity" => "$local"
        }},
        "jdbc-realm" => undefined,
        "kerberos-security-factory" => undefined,
        "key-managers" => undefined,
        "key-store" => undefined,
        "key-store-realm" => undefined,
        "ldap-key-store" => undefined,
        "ldap-realm" => undefined,
        "logical-permission-mapper" => undefined,
        "logical-role-mapper" => undefined,
        "mapped-regex-realm-mapper" => undefined,
        "mechanism-provider-filtering-sasl-server-factory" => {"elytron" => {
            "enabling" => true,
            "filters" => [{"provider-name" => "WildFlyElytron"}],
            "sasl-server-factory" => "global"
        }},
        "properties-realm" => {
            "ApplicationRealm" => {
                "groups-attribute" => "groups",
                "groups-properties" => {
                    "path" => "application-roles.properties",
                    "relative-to" => "jboss.server.config.dir"
```

```
                },
                "users-properties" => {
                    "path" => "application-users.properties",
                    "relative-to" => "jboss.server.config.dir",
                    "digest-realm-name" => "ApplicationRealm"
                }
            },
            "ManagementRealm" => {
                "groups-attribute" => "groups",
                "groups-properties" => {
                    "path" => "mgmt-groups.properties",
                    "relative-to" => "jboss.server.config.dir"
                },
                "users-properties" => {
                    "path" => "mgmt-users.properties",
                    "relative-to" => "jboss.server.config.dir",
                    "digest-realm-name" => "ManagementRealm"
                }
            }
        },
        "provider-http-server-mechanism-factory" => {"global" => {"providers" => undefined}},
        "provider-loader" => {
            "elytron" => {
                "class-names" => undefined,
                "configuration" => undefined,
                "module" => "org.wildfly.security.elytron",
                "path" => undefined,
                "relative-to" => undefined
            },
            "openssl" => {
                "class-names" => undefined,
                "configuration" => undefined,
                "module" => "org.wildfly.openssl",
                "path" => undefined,
                "relative-to" => undefined
            }
        },
        "provider-sasl-server-factory" => {"global" => {"providers" => undefined}},
        "regex-principal-transformer" => undefined,
        "regex-validating-principal-transformer" => undefined,
        "sasl-authentication-factory" => {
            "management-sasl-authentication" => {
                "mechanism-configurations" => [
                    {
                        "mechanism-name" => "JBOSS-LOCAL-USER",
                        "realm-mapper" => "local"
                    },
                    {
                        "mechanism-name" => "DIGEST-MD5",
                        "mechanism-realm-configurations" => [{"realm-name" =>
"ManagementRealm"}]
                    }
                ],
                "sasl-server-factory" => "configured",
                "security-domain" => "ManagementDomain"
            },
            "application-sasl-authentication" => {
                "mechanism-configurations" => undefined,
```

```
                "sasl-server-factory" => "configured",
                "security-domain" => "ApplicationDomain"
            }
        },
        "security-domain" => {
            "ApplicationDomain" => {
                "default-realm" => "ApplicationRealm",
                "permission-mapper" => "default-permission-mapper",
                "post-realm-principal-transformer" => undefined,
                "pre-realm-principal-transformer" => undefined,
                "principal-decoder" => undefined,
                "realm-mapper" => undefined,
                "realms" => [{
                    "realm" => "ApplicationRealm",
                    "role-decoder" => "groups-to-roles"
                }],
                "role-mapper" => undefined,
                "trusted-security-domains" => undefined
            },
            "ManagementDomain" => {
                "default-realm" => "ManagementRealm",
                "permission-mapper" => "default-permission-mapper",
                "post-realm-principal-transformer" => undefined,
                "pre-realm-principal-transformer" => undefined,
                "principal-decoder" => undefined,
                "realm-mapper" => undefined,
                "realms" => [
                    {
                        "realm" => "ManagementRealm",
                        "role-decoder" => "groups-to-roles"
                    },
                    {
                        "realm" => "local",
                        "role-mapper" => "super-user-mapper"
                    }
                ],
                "role-mapper" => undefined,
                "trusted-security-domains" => undefined
            }
        },
        "security-property" => undefined,
        "server-ssl-context" => undefined,
        "service-loader-http-server-mechanism-factory" => undefined,
        "service-loader-sasl-server-factory" => undefined,
        "simple-permission-mapper" => undefined,
        "simple-regex-realm-mapper" => undefined,
        "simple-role-decoder" => {"groups-to-roles" => {"attribute" => "groups"}},
        "token-realm" => undefined,
        "trust-managers" => undefined,
        "x500-attribute-principal-decoder" => undefined
    }
}
```

15.9.4 Default Application Authentication Configuration

By default, applications are secured using legacy security domains. Applications must specify a security domain in their *web.xml* as well as the authentication method. If no security domain is specified by the application, WildFly will use the provided *other* legacy security domain.

Update WildFly to Use the Default Elytron Components for Application Authentication

```
/subsystem=undertow/application-security-domain=exampleApplicationDomain:add(http-authentication-f
```

Default Elytron Application HTTP Authentication Configuration

By default, the *application-http-authentication* http-authentication-factory is provided for application http authentication.

```
/subsystem=elytron/http-authentication-factory=application-http-authentication:read-resource()
{
    "outcome" => "success",
    "result" => {
        "http-server-mechanism-factory" => "global",
        "mechanism-configurations" => [
            {
                "mechanism-name" => "BASIC",
                "mechanism-realm-configurations" => [{"realm-name" => "Application Realm"}]
            },
            {"mechanism-name" => "FORM"}
        ],
        "security-domain" => "ApplicationDomain"
    }
}
```

The *application-http-authentication* http-authentication-factory is configured to use the *ApplicationDomain* security domain.

```
/subsystem=elytron/security-domain=ApplicationDomain:read-resource()
{
    "outcome" => "success",
    "result" => {
        "default-realm" => "ApplicationRealm",
        "permission-mapper" => "default-permission-mapper",
        "post-realm-principal-transformer" => undefined,
        "pre-realm-principal-transformer" => undefined,
        "principal-decoder" => undefined,
        "realm-mapper" => undefined,
        "realms" => [{
            "realm" => "ApplicationRealm",
            "role-decoder" => "groups-to-roles"
        }],
        "role-mapper" => undefined,
        "trusted-security-domains" => undefined
    }
}
```

The *ApplicationDomain* security domain is backed by the *ApplicationRealm* Elytron security realm, which is a properties-based realm.

```
/subsystem=elytron/properties-realm=ApplicationRealm:read-resource()
{
    "outcome" => "success",
    "result" => {
        "groups-attribute" => "groups",
        "groups-properties" => {
            "path" => "application-roles.properties",
            "relative-to" => "jboss.server.config.dir"
        },
        "users-properties" => {
            "path" => "application-users.properties",
            "relative-to" => "jboss.server.config.dir",
            "digest-realm-name" => "ApplicationRealm"
        }
    }
}
```

15.9.5 Default Management Authentication Configuration

By default, the WildFly management interfaces are secured by the legacy core management authentication.

Default Configuration

```
/core-service=management/management-interface=http-interface:read-resource()
{
    "outcome" => "success",
    "result" => {
        "allowed-origins" => undefined,
        "console-enabled" => true,
        "http-authentication-factory" => undefined,
        "http-upgrade" => {"enabled" => true},
        "http-upgrade-enabled" => true,
        "sasl-protocol" => "remote",
        "secure-socket-binding" => undefined,
        "security-realm" => "ManagementRealm",
        "server-name" => undefined,
        "socket-binding" => "management-http",
        "ssl-context" => undefined
    }
```

WildFly does provide *management-http-authentication* and *management-sasl-authentication* in the *elytron* subsystem for securing the management interfaces as well.

Update WildFly to Use the Default Elytron Components for Management Authentication

Set http-authentication-factory to use management-http-authentication

```
/core-service=management/management-interface=http-interface:write-attribute( \
  name=http-authentication-factory, \
  value=management-http-authentication \
)
```

Set sasl-authentication-factory to use management-sasl-authentication

```
/core-service=management/management-interface=http-interface:write-attribute( \
  name=http-upgrade.sasl-authentication-factory, \
  value=management-sasl-authentication \
)
```

Undefine security-realm

```
/core-service=management/management-interface=http-interface:undefine-attribute(name=security-realm
```

Reload WildFly for the changes to take affect.

```
reload
```

The management interfaces are now secured using the default components provided by the 'elytron' subsystem.

Default Elytron Management HTTP Authentication Configuration

When you access the management interface over HTTP, for example when using the web-based management console, WildFly will use the *management-http-authentication* http-authentication-factory.

```
/subsystem=elytron/http-authentication-factory=management-http-authentication:read-resource()
{
    "outcome" => "success",
    "result" => {
        "http-server-mechanism-factory" => "global",
        "mechanism-configurations" => [{
            "mechanism-name" => "DIGEST",
            "mechanism-realm-configurations" => [{"realm-name" => "ManagementRealm"}]
        }],
        "security-domain" => "ManagementDomain"
    }
}
```

The *management-http-authentication* http-authentication-factory, is configured to use the *ManagementDomain* security domain.

```
/subsystem=elytron/security-domain=ManagementDomain:read-resource()
{
    "outcome" => "success",
    "result" => {
        "default-realm" => "ManagementRealm",
        "permission-mapper" => "default-permission-mapper",
        "post-realm-principal-transformer" => undefined,
        "pre-realm-principal-transformer" => undefined,
        "principal-decoder" => undefined,
        "realm-mapper" => undefined,
        "realms" => [
            {
                "realm" => "ManagementRealm",
                "role-decoder" => "groups-to-roles"
            },
            {
                "realm" => "local",
                "role-mapper" => "super-user-mapper"
            }
        ],
        "role-mapper" => undefined,
        "trusted-security-domains" => undefined
    }
}
```

The *ManagementDomain* security domain is backed by the *ManagementRealm* Elytron security realm, which is a properties-based realm.

```
/subsystem=elytron/properties-realm=ManagementRealm:read-resource()
{
    "outcome" => "success",
    "result" => {
        "groups-attribute" => "groups",
        "groups-properties" => {
            "path" => "mgmt-groups.properties",
            "relative-to" => "jboss.server.config.dir"
        },
        "plain-text" => false,
        "users-properties" => {
            "path" => "mgmt-users.properties",
            "relative-to" => "jboss.server.config.dir"
        }
    }
}
```

Default Elytron Management CLI Authentication

By default, the management CLI (*jboss-cli.sh*) is configured to connect over *remotehttp*.

Default jboss-cli.xml

```
<jboss-cli xmlns="urn:jboss:cli:3.1">

    <default-protocol use-legacy-override="true">remotehttp</default-protocol>

    <!-- The default controller to connect to when 'connect' command is executed w/o arguments
-->
    <default-controller>
        <protocol>remotehttp</protocol>
        <host>localhost</host>
        <port>9990</port>
    </default-controller>
```

This will establish a connection over HTTP and use HTTP upgrade to change the communication protocol to *native*. The HTTP upgrade connection is secured in the *http-upgrade* section of the *http-interface* using a *sasl-authentication-factory*.

Example Configuration with Default Components

```
/core-service=management/management-interface=http-interface:read-resource()
{
    "outcome" => "success",
    "result" => {
        "allowed-origins" => undefined,
        "console-enabled" => true,
        "http-authentication-factory" => "management-http-authentication",
        "http-upgrade" => {
            "enabled" => true,
            "sasl-authentication-factory" => "management-sasl-authentication"
        },
        "http-upgrade-enabled" => true,
        "sasl-protocol" => "remote",
        "secure-socket-binding" => undefined,
        "security-realm" => undefined,
        "server-name" => undefined,
        "socket-binding" => "management-http",
        "ssl-context" => undefined
    }
}
```

The default sasl-authentication-factory is *management-sasl-authentication*.

```
/subsystem=elytron/sasl-authentication-factory=management-sasl-authentication:read-resource()
{
    "outcome" => "success",
    "result" => {
        "mechanism-configurations" => [
            {
                "mechanism-name" => "JBOSS-LOCAL-USER",
                "realm-mapper" => "local"
            },
            {
                "mechanism-name" => "DIGEST-MD5",
                "mechanism-realm-configurations" => [{"realm-name" => "ManagementRealm"}]
            }
        ],
        "sasl-server-factory" => "configured",
        "security-domain" => "ManagementDomain"
    }
}
```

The *management-sasl-authentication* sasl-authentication-factory specifies *JBOSS-LOCAL-USER* and *DIGEST-MD5* mechanisms.

JBOSS-LOCAL-USER Realm

```
/subsystem=elytron/identity-realm=local:read-resource()
{
    "outcome" => "success",
    "result" => {
        "attribute-name" => undefined,
        "attribute-values" => undefined,
        "identity" => "$local"
    }
}
```

The *local* Elytron security realm is for handling silent authentication for local users.

The *ManagementRealm* Elytron security realm is the same realm used in the *management-http-authentication* http-authentication-factory.

15.9.6 Comparing Legacy Approaches to Elytron Approaches

Legacy Approach	Elytron Approach
UsersRoles Login Module	Configure Authentication with a Properties File-Based Identity Store
Database Login Module	Configure Authentication with a Database Identity Store
Ldap, LdapExtended, AdvancedLdap, AdvancedADLdap Login Modules	Configure Authentication with an LDAP-Based Identity Store
Certificate, Certificate Roles Login Module	Configure Authentication with Certificates
Kerberos, SPNEGO Login Modules	Configure Authentication with a Kerberos-Based Identity Store
Kerberos, SPNEGO Login Modules with Fallback	Configure Authentication with a Form as a Fallback for Kerberos
Vault	Create and Use a Credential Store
Legacy Security Realms	Secure the Management Interfaces with a New Identity Store, Silent Authentication
RBAC	Using RBAC with Elytron
Legacy Security Realms for One-way and Two-way SSL/TLS for Applications	Enable One-way SSL/TLS for Applications, Enable Two-way SSL/TLS in WildFly for Applications
Legacy Security Realms for One-way and Two-way SSL/TLS for Management Interfaces	Enable One-way for the Management Interfaces Using the Elytron Subsystem, Enable Two-way SSL/TLS for the Management Interfaces using the Elytron Subsystem

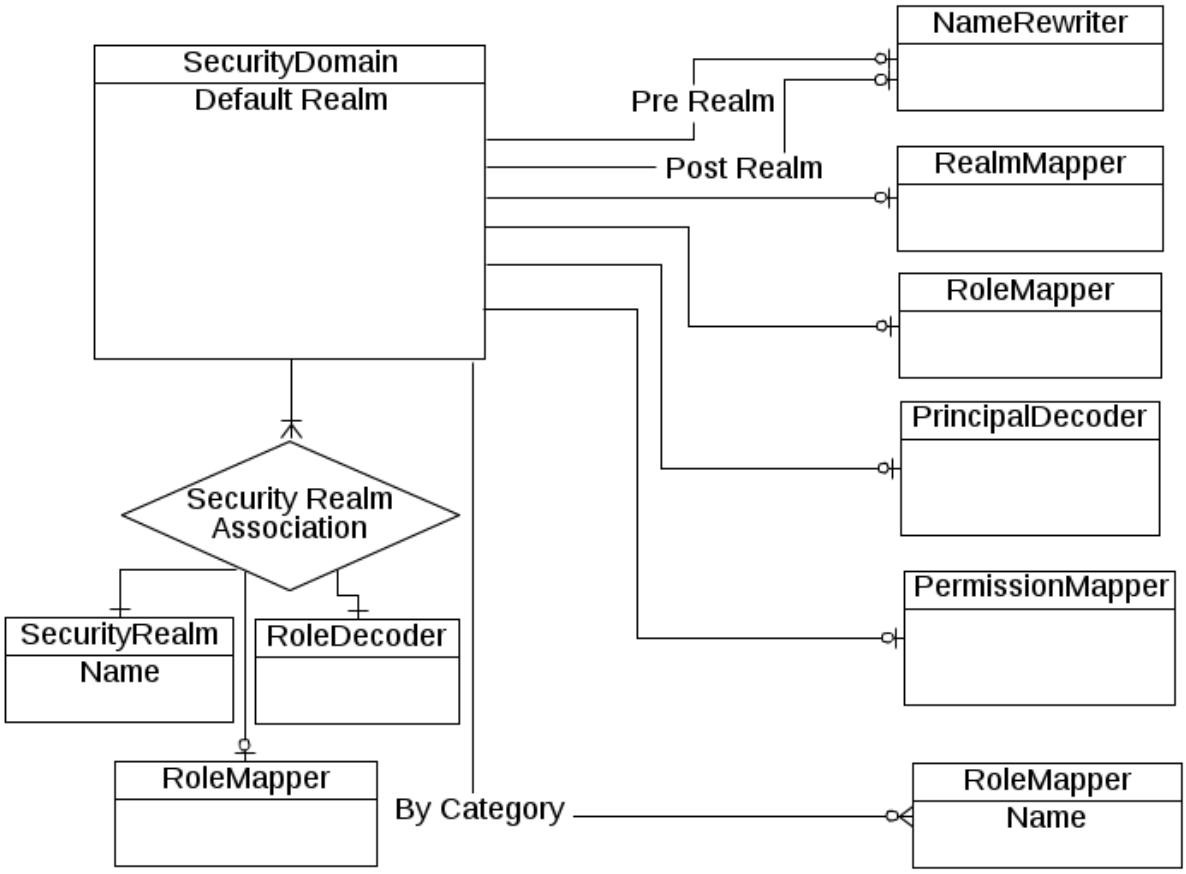
15.10 General Elytron Architecture

The overall architecture for WildFly Elytron is building up a full security policy from assembling smaller components together, by default we include various implementations of the components - in addition to this, custom implementations of many components can be provided in order to provide more specialised implementations.

Within WildFly the different Elytron components are handled as capabilities meaning that different implementations can be mixed and matched, however the different implementations are modelled using distinct resources. This section contains a number of diagrams to show the general relationships between different components to provide a high level view, however the different resource definitions may use different dependencies depending on their purpose.

15.10.1 Security Domains

Within WildFly Elytron a SecurityDomain can be considered as a security policy backed by one or more SecurityRealm instances. Resources that make authorization decisions will be associated with a SecurityDomain, from the SecurityDomain a SecurityIdentity can be obtained which is a representation of the current identity, from this the identities roles and permissions can be checked to make the authorization decision for the resource.



SecurityDomain
The SecurityDomain is the general wrapper around the policy describing a resulting SecurityIdentity and makes use of the following components to define this policy.

- NameRewriter

NameRewriters are used in multiple places within the Elytron configuration, as their name implies, their purpose is to take a name and map it to another representation of the name or perform some normalisation or clean up of the name.

- RealmMapper

As a SecurityDomain is able to reference multiple SecurityRealms the RealmMapper is responsible for identifying which SecurityRealm to use based on the supplied name for authentication.

- SecurityRealm

One more more named SecurityRealms are associated with a SecurityDomain, the SecurityRealms are the access to the underlying repository of identities and are used for obtaining credentials to allow authentication mechanisms to perform verification, for validation of Evidence and for obtaining the raw AuthorizationIdentity performing the authentication.

Some SecurityRealm implementations are also modifiable so expose an API that allows for updates to be made to the repository containing the identities.

- RoleDecoder

Along with the SecurityRealm association is also a reference to a RoleDecoder, the RoleDecoder takes the raw AuthorizationIdentity returned from the SecurityRealm and converts it's attributes into roles.

- RoleMapper

After the roles have been decoded for an identity further mapping can be applied, this could be as simple at normalising the format of the names through to adding or removing specific role names. If a RoleMapper is referenced by the SecurityRealm association that RoleMapper is applied first before applying the RoleMapper associated with the SecurityDomain.

- PrincipalDecoder

A PrincipalDecoder converts from a Principal to a String representation of a name, one example for this is we have an X500PrincipalDecoder which is able to extract an attribute from a distinguished name.

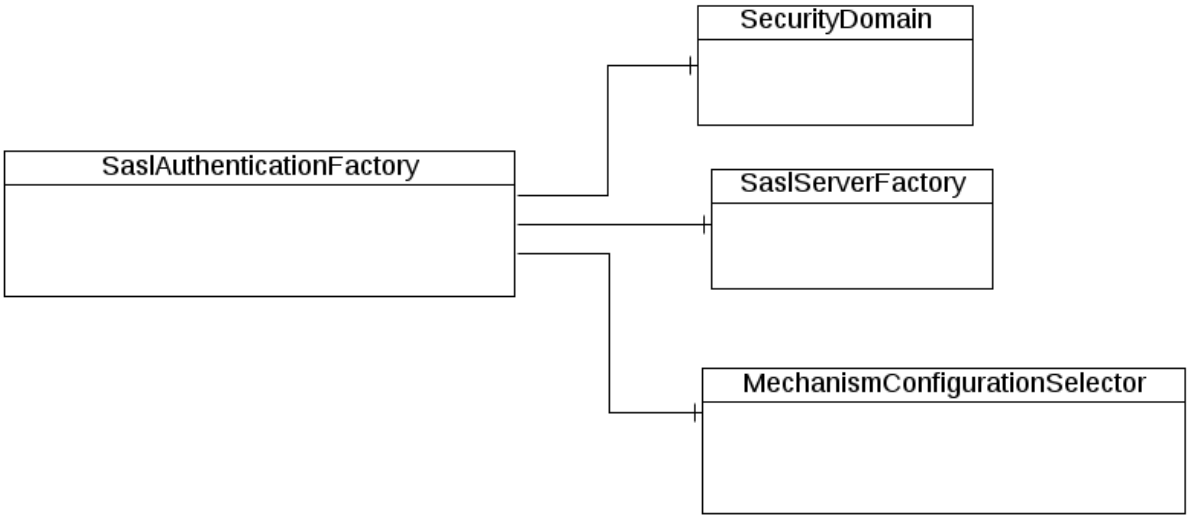
- PermissionMapper

In addition to having roles a SecurityIdentity can also have a set of permissions, the PermissionMapper assigns those permissions to the identity.

Different secured resources can be associated with different SecurityDomains for their authorization decisions, within WildFly Elytron we have the ability to configure inflow between different SecurityDomains. The inflow process means that a SecurityIdentity inflowed into a second SecurityDomain has the mappings of the new SecurityDomain applied to it so although a common identity may be calling different resources each of those resources could have a very different view.

15.10.2 SASL Authentication

The SaslAuthenticationFactory is an authentication policy for authentication using SASL authentication mechanisms, in addition to being a policy it is also a factory for configured authentication mechanisms backed by a SecurityDomain.



SaslAuthenticationFactory
The SaslAuthenticationFactory references the following: -

- SecurityDomain

This is the security domain that any mechanism authentication will be performed against.

- SaslServerFactory

This is the general factory for server side SASL authentication mechanisms.

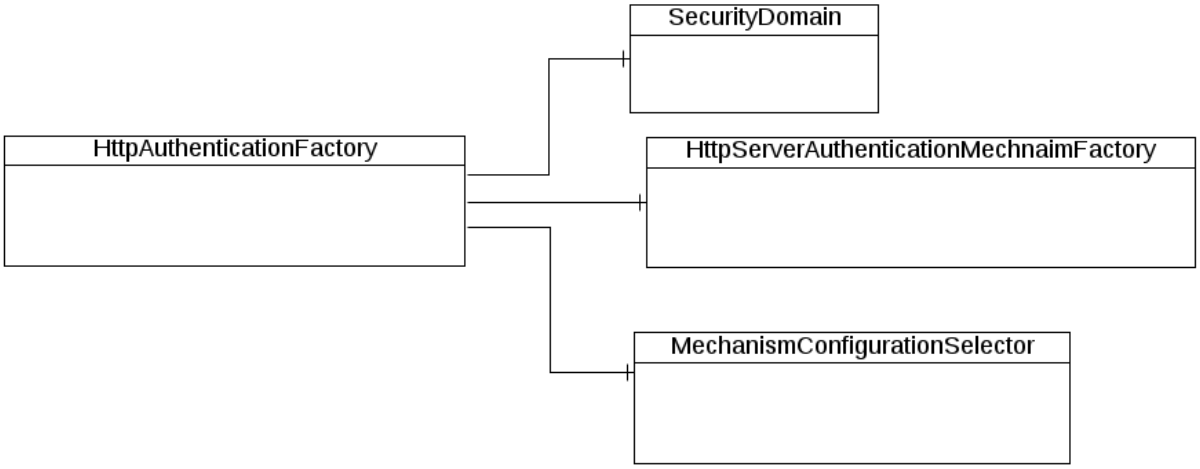
- MechanismConfigurationSelector

Additional configuration can be supplied for the authentication mechanisms, the configuration will be described in more detail later but the purpose of the MechanismConfigurationSelector is to obtain configuration specific to the mechanism selected. This can include information about realm names a mechanism should present to a remote client plus additional NameRewriters and RealmMappers to use during the authentication process.

The reason some of the components referenced by the SecurityDomain are duplicated is so that mechanism specific mappings can be applied.

15.10.3 HTTP Authentication

The HttpAuthenticationFactory is an authentication policy for authentication using HTTP authentication mechanisms, in addition to being a policy it is also a factory for configured authentication mechanisms backed by a SecurityDomain.



HttpAuthenticationFactory
The HttpAuthenticationFactory references the following: -

- SecurityDomain

This is the security domain that any mechanism authentication will be performed against.

- HttpServerAuthenticationMechanismFactory

This is the general factory for server side HTTP authentication mechanisms.

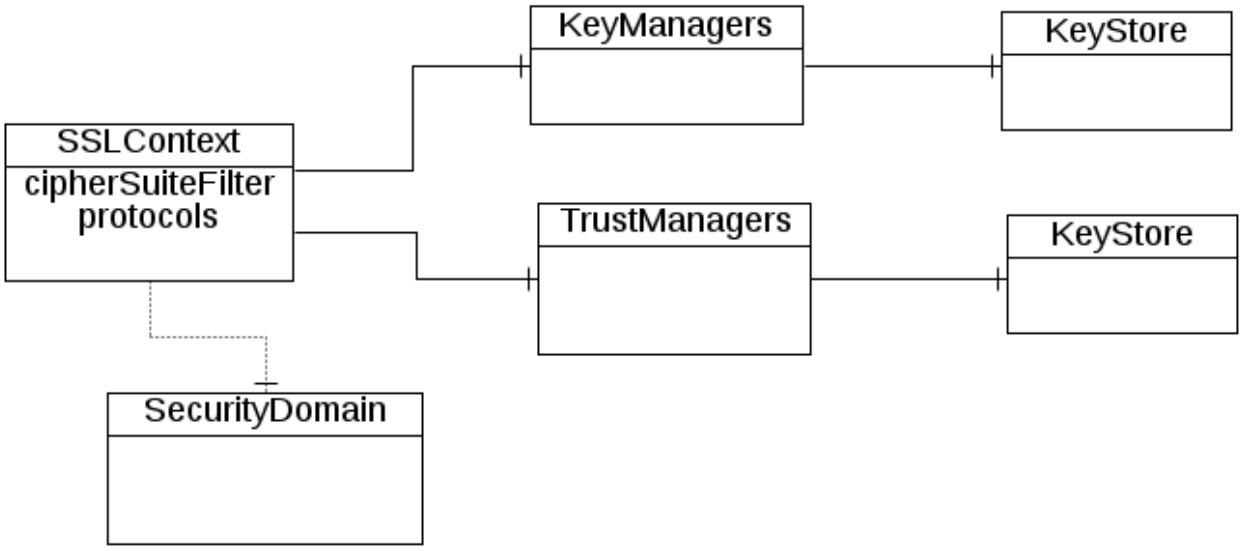
- MechanismConfigurationSelector

Additional configuration can be supplied for the authentication mechanisms, the configuration will be described in more detail later but the purpose of the MechanismConfigurationSelector is to obtain configuration specific to the mechanism selected. This can include information about realm names a mechanism should present to a remote client plus additional NameRewriters and RealmMappers to use during the authentication process.

The reason some of the components referenced by the SecurityDomain are duplicated is so that mechanism specific mappings can be applied.

15.10.4 SSL / TLS

The SSLContext defined within Elytron is a javax.net.ssl.SSLContext meaning it can be used by anything that uses an SSLContext directly.



SSLContext
In addition to the usual configuration for an SSLContext it is possible to configure additional items such as cipher suites and protocols and the SSLContext returned will wrap any engines created to set these values.

The SSLContext within Elytron can also reference the following: -

- KeyManagers

An array of KeyManager instances to be used by the SSLContext, this in turn can reference a KeyStore to load the keys.

- TrustManagers

An array of TrustManager instances to be used by the SSLContext, this in turn can also reference a KeyStore to load the certificates.

- SecurityDomain

This is optional, however if an SSLContext is configured to reference a SecurityDomain then verification of a clients certificate can be performed as an authentication ensuring the appropriate permissions to Logon are assigned before even allowing the connection to be fully opened, additionally the SecurityIdentity can be established at the time the connection is opened and used for any invocations over the connection.

15.11 Migrate Legacy Security to Elytron Security

15.11.1 Authentication Configuration

Properties Based Authentication / Authorization

PicketBox Based Configuration

This migration example assumes a deployed web application is configured to require authentication using FORM based authentication and is referencing a PicketBox based security domain using the UsersRolesLoginModule to load user information from a pair or properties files.

Original Configuration

A security domain can be defined in the legacy security subsystem using the following management operations: -

```
./subsystem=security/security-domain=application-security:add
./subsystem=security/security-domain=application-security/authentication=classic:add(login-modules=
flag=Required,
module-options={usersProperties=file://${jboss.server.config.dir}/example-users.properties,
rolesProperties=file://${jboss.server.config.dir}/example-roles.properties}}])
```

This would result in a security domain definition: -

```
<security-domain name="application-security">
    <authentication>
      <login-module code="UsersRoles" flag="required">
        <module-option name="usersProperties"
value="file://${jboss.server.config.dir}/example-users.properties"/>
        <module-option name="rolesProperties"
value="file://${jboss.server.config.dir}/example-roles.properties"/>
      </login-module>
    </authentication>
  </security-domain>
```

Intermediate Configuration

It is possible to take a previously defined PicketBox security domain and expose it as an Elytron security realm so it can be wired into a complete Elytron based configuration, if only properties based authentication was to be migrated it would be recommended to jump to the fully migration configuration and avoid the unnecessary dependency on the legacy security subsystem but for situations where that is not immediately possible these commands illustrate an intermediate solution.

These steps assume the original configuration is already in place.

The first step is to add a mapping to an Elytron security realm within the legacy security subsystem.

```
./subsystem=security/elytron-realm=application-security:add(legacy-jaas-config=application-securit
```

This results in the following configuration.

```
<subsystem xmlns="urn:jboss:domain:security:2.0">
    ...
    <elytron-integration>
      <security-realms>
        <elytron-realm name="application-security" legacy-jaas-config="application-security"/>
      </security-realms>
    </elytron-integration>
    ...
  </subsystem>
```

Within the Elytron subsystem a security domain can be defined which references the exported security realm and also a http authentication factory which supports FORM based authentication.

```
./subsystem=elytron/security-domain=application-security:add(realms=[{realm=application-security}]
default-realm=application-security, permission-mapper=default-permission-mapper)
./subsystem=elytron/http-authentication-factory=application-security-http:add(http-server-mechanis
security-domain=application-security, mechanism-configurations=[{mechanism-name=FORM}])
```

And the resulting configuration: -

```
<subsystem xmlns="urn:wildfly:elytron:1.0" final-providers="combined-providers"
disallowed-providers="OracleUcrypto">
    ...
    <security-domains>
      ...
      <security-domain name="application-security" default-realm="application-security"
permission-mapper="default-permission-mapper">
        <realm name="application-security"/>
      </security-domain>
    </security-domains>
    ...
    <http>
      ...
      <http-authentication-factory name="application-security-http"
http-server-mechanism-factory="global" security-domain="application-security">
        <mechanism-configuration>
          <mechanism mechanism-name="FORM"/>
        </mechanism-configuration>
      </http-authentication-factory>
      ...
    </http>
    ...
  </subsystem>
```

Finally configuration needs to be added to the Undertow subsystem to map the security domain referenced by the deployment to the newly defined http authentication factory.

```
./subsystem=undertow/application-security-domain=application-security:add(http-authentication-fact
```

Which results in: -

```
<subsystem xmlns="urn:jboss:domain:undertow:4.0">
    ...
    <application-security-domains>
      <application-security-domain name="application-security"
http-authentication-factory="application-security-http"/>
    </application-security-domains>
    ...
  </subsystem>
```

Note: If the deployment was already deployed at this point the application server should be reloaded or the deployment redeployed for the application security domain mapping to take effect.

The following command can then be used to verify the mapping was applied to the deployment.

```
[standalone@localhost:9990 /]
./subsystem=undertow/application-security-domain=application-security:read-resource(include-runtim
"outcome" => "success",
    "result" => {
        "enable-jacc" => false,
        "http-authentication-factory" => "application-security-http",
        "override-deployment-config" => false,
        "referencing-deployments" => ["HelloWorld.war"],
        "setting" => undefined
    }
}
```

The deployment being tested here is 'HelloWorld.war' and the output from the previous command shows this deployment is referencing the mapping.

At this stage the previously defined security domain is used for it's LoginModule configuration but this is wrapped by Elytron components which take over authentication.

Fully Migrated Configuration

Alternatively the configuration can be completely defined within the Elytron subsystem, in this case it is assumed none of the previous commands have been executed and this is started from a clean configuration - however if the security domain definition does exist in the legacy security subsystem that will remain completely independent.

First a new security realm can be defined within the Elytron subsystem referencing the files referenced previously: -

```
./subsystem=elytron/properties-realm=application-properties:add(users-properties={path=example-use
relative-to=jboss.server.config.dir, plain-text=true, digest-realm-name="Application Security"},
groups-properties={path=example-roles.properties, relative-to=jboss.server.config.dir},
groups-attribute=Roles)
```

As before a security domain and http authentication factory can be defined.

```
./subsystem=elytron/security-domain=application-security:add(realms=[{realm=application-properties
default-realm=application-properties, permission-mapper=default-permission-mapper)
./subsystem=elytron/http-authentication-factory=application-security-http:add(http-server-mechanis
security-domain=application-security, mechanism-configurations=[{mechanism-name=FORM}])
```

This results in the following overall configuration.

```
<subsystem xmlns="urn:wildfly:elytron:1.0" final-providers="combined-providers"
disallowed-providers="OracleUcrypto">
    ...
    <security-domains>
      ...
      <security-domain name="application-security" default-realm="application-properties"
permission-mapper="default-permission-mapper">
        <realm name="application-properties"/>
      </security-domain>
    </security-domains>
    <security-realms>
      ...
      <properties-realm name="application-properties" groups-attribute="Roles">
        <users-properties path="example-users.properties" relative-to="jboss.server.config.dir"
digest-realm-name="Application Security" plain-text="true"/>
        <groups-properties path="example-roles.properties"
relative-to="jboss.server.config.dir"/>
      </properties-realm>
    </security-realms>
    ...
    <http>
      ...
      <http-authentication-factory name="application-security-http"
http-server-mechanism-factory="global" security-domain="application-security">
        <mechanism-configuration>
          <mechanism mechanism-name="FORM"/>
        </mechanism-configuration>
      </http-authentication-factory>
      ...
    </http>
    ...
  </subsystem>
```

As before the application-security-domain mapping should be added to the Undertow subsystem and the server reloaded or the deployment redeployed as required.

```
./subsystem=undertow/application-security-domain=application-security:add(http-authentication-fact
```

Which results in: -

```
<subsystem xmlns="urn:jboss:domain:undertow:4.0">
    ...
    <application-security-domains>
      <application-security-domain name="application-security"
http-authentication-factory="application-security-http"/>
    </application-security-domains>
    ...
  </subsystem>
```

At this stage the authentication is the equivalent of the original configuration however now Elytron components are used exclusively.

Legacy Security Realm

Original Configuration

A legacy security realm can be defined using the following commands to load users passwords and group information from properties files.

```
./core-service=management/security-realm=ApplicationSecurity:add
./core-service=management/security-realm=ApplicationSecurity/authentication=properties:add(relativ
path=example-users.properties, plain-text=true)
./core-service=management/security-realm=ApplicationSecurity/authorization=properties:add(relative
path=example-roles.properties)
```

This results in the following realm definition.

```
<security-realm name="ApplicationSecurity">
    <authentication>
      <properties path="example-users.properties" relative-to="jboss.server.config.dir"
plain-text="true"/>
    </authentication>
    <authorization>
      <properties path="example-roles.properties" relative-to="jboss.server.config.dir"/>
    </authorization>
  </security-realm>
```

A legacy security realm would typically be used to secure either the management interfaces or remoting connectors.

Migrated Configuration

One of the motivations for adding the Elytron based security to the application server is to allow a consistent security solution to be used across the server, to replace the security realm the same steps as described in the previous 'Fully Migrated' section can be followed again up until the http-authentication-factory is defined.

A legacy security realm can also be used for SASL based authentication so a sasl-authentication-factory should also be defined.

```
./subsystem=elytron/sasl-authentication-factory=application-security-sasl:add(sasl-server-factory=
security-domain=application-security, mechanism-configurations=[{mechanism-name=PLAIN}])
```

```
<subsystem xmlns="urn:wildfly:elytron:1.0" final-providers="combined-providers"
disallowed-providers="OracleUcrypto">
    ...
    <sasl>
      ...
      <sasl-authentication-factory name="application-security-sasl"
sasl-server-factory="elytron" security-domain="application-security">
        <mechanism-configuration>
          <mechanism mechanism-name="PLAIN"/>
        </mechanism-configuration>
      </sasl-authentication-factory>
      ...
    </sasl>
  </subsystem>
```

This can be associated with a Remoting connector to use for authentication and the existing security realm reference cleared.

```
./subsystem=remoting/http-connector=http-remoting-connector:write-attribute(name=sasl-authenticati
value=application-security-sasl)
./subsystem=remoting/http-connector=http-remoting-connector:undefine-attribute(name=security-realm
```

```
<subsystem xmlns="urn:jboss:domain:remoting:4.0">
    ...
    <http-connector name="http-remoting-connector" connector-ref="default"
sasl-authentication-factory="application-security-sasl"/>
  </subsystem>
```

If this new configuration was to be used to secure the management interfaces more suitable names should be chosen but the following commands illustrate how to set the two authentication factories and clear the existing security realm reference.

```
./core-service=management/management-interface=http-interface:write-attribute(name=http-authentica
value=application-security-http)
./core-service=management/management-interface=http-interface:write-attribute(name=http-upgrade.sa
value=application-security-sasl)
./core-service=management/management-interface=http-interface:undefine-attribute(name=security-rea
```

```
<management-interfaces>
    <http-interface http-authentication-factory="application-security-http">
      <http-upgrade enabled="true" sasl-authentication-factory="application-security-sasl"/>
      <socket-binding http="management-http"/>
    </http-interface>
  </management-interfaces>
```

LDAP Authentication Migration

The section describing how to migrate from properties based authentication using either PicketBox or legacy security realms to Elytron also contained a lot of additional information regarding defining security domains, authentication factories, and how these are mapped to be used for authentication. This section will illustrate some equivalent LDAP configuration using legacy security realms and PicketBox security domains and show the equivalent configuration using Elytron but will not repeat the steps to wire it all together covered in the previous section.

These configuration examples are developed against a test LDAP sever with user entries like: -

```
dn: uid=TestUserOne,ou=users,dc=group-to-principal,dc=wildfly,dc=org
objectClass: top
objectClass: inetOrgPerson
objectClass: uidObject
objectClass: person
objectClass: organizationalPerson
cn: Test User One
sn: Test User One
uid: TestUserOne
userPassword: {SSHA}UG8ov2rnrnBKakcARVvraZHqTa7mFWJZlWt2HA==
```

The group entries then look like: -

```
dn: uid=GroupOne,ou=groups,dc=group-to-principal,dc=wildfly,dc=org
objectClass: top
objectClass: groupOfUniqueNames
objectClass: uidObject
cn: Group One
uid: GroupOne
uniqueMember: uid=TestUserOne,ou=users,dc=group-to-principal,dc=wildfly,dc=org
```

For authentication purposes the username will be matched against the 'uid' attribute, also the resulting group name will be taken from the 'uid' attribute of the group entry.

Legacy Security Realm

A connection to the LDAP server and related security realm can be created with the following commands: -

```
batch
./core-service=management/ldap-connection=MyLdapConnection:add(url="ldap://localhost:10389",
search-dn="uid=admin,ou=system", search-credential="secret")

./core-service=management/security-realm=LDAPRealm:add
./core-service=management/security-realm=LDAPRealm/authentication=ldap:add(connection="MyLdapConne
username-attribute=uid, base-dn="ou=users,dc=group-to-principal,dc=wildfly,dc=org")


./core-service=management/security-realm=LDAPRealm/authorization=ldap:add(connection=MyLdapConnect
base-dn="ou=users,dc=group-to-principal,dc=wildfly,dc=org")
./core-service=management/security-realm=LDAPRealm/authorization=ldap/group-search=group-to-princi
iterative=true, prefer-original-connection=true, principal-attribute=uniqueMember,
search-by=DISTINGUISHED_NAME, group-name=SIMPLE, group-name-attribute=uid)
run-batch
```

This results in the following configuration.

```
<management>
    <security-realms>
      ...
      <security-realm name="LDAPRealm">
        <authentication>
          <ldap connection="MyLdapConnection"
base-dn="ou=users,dc=group-to-principal,dc=wildfly,dc=org">
            <username-filter attribute="uid"/>
          </ldap>
        </authentication>
        <authorization>
          <ldap connection="MyLdapConnection">
            <username-to-dn>
              <username-filter base-dn="ou=users,dc=group-to-principal,dc=wildfly,dc=org"
attribute="uid"/>
            </username-to-dn>
            <group-search group-name="SIMPLE" iterative="true" group-name-attribute="uid">
              <group-to-principal search-by="DISTINGUISHED_NAME"
base-dn="ou=groups,dc=group-to-principal,dc=wildfly,dc=org" prefer-original-connection="true">
                <membership-filter principal-attribute="uniqueMember"/>
              </group-to-principal>
            </group-search>
          </ldap>
        </authorization>
      </security-realm>
    </security-realms>
    <outbound-connections>
      <ldap name="MyLdapConnection" url="ldap://localhost:10389" search-dn="uid=admin,ou=system"
search-credential="secret"/>
    </outbound-connections>
    ...
  </management>
```

PicketBox LdapExtLoginModule

The following commands can create a PicketBox security domain configured to use the LdapExtLoginModule to verify a username and password.

```
./subsystem=security/security-domain=application-security:add
./subsystem=security/security-domain=application-security/authentication=classic:add(login-modules=
flag=Required, module-options={ \
java.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory, \
java.naming.provider.url=ldap://localhost:10389, \
java.naming.security.authentication=simple, \
bindDN="uid=admin,ou=system", \
bindCredential=secret, \
baseCtxDN="ou=users,dc=group-to-principal,dc=wildfly,dc=org", \
baseFilter="(uid={0})", \
rolesCtxDN="ou=groups,dc=group-to-principal,dc=wildfly,dc=org",\
roleFilter="(uniqueMember={1})", \
roleAttributeID="uid" \
}])
```

This results in the following configuration.

```
<subsystem xmlns="urn:jboss:domain:security:2.0">
    ...
    <security-domains>
      ...
      <security-domain name="application-security">
        <authentication>
          <login-module code="LdapExtended" flag="required">
            <module-option name="java.naming.factory.initial"
value="com.sun.jndi.ldap.LdapCtxFactory"/>
            <module-option name="java.naming.provider.url" value="ldap://localhost:10389"/>
            <module-option name="java.naming.security.authentication" value="simple"/>
            <module-option name="bindDN" value="uid=admin,ou=system"/>
            <module-option name="bindCredential" value="secret"/>
            <module-option name="baseCtxDN"
value="ou=users,dc=group-to-principal,dc=wildfly,dc=org"/>
            <module-option name="baseFilter" value="(uid={0})"/>
            <module-option name="rolesCtxDN"
value="ou=groups,dc=group-to-principal,dc=wildfly,dc=org"/>
            <module-option name="roleFilter" value="(uniqueMember={1})"/>
            <module-option name="roleAttributeID" value="uid"/>
          </login-module>
        </authentication>
      </security-domain>
    </security-domains>
  </subsystem>
```

Migrated

Within the Elytron subsystem a directory context can be defined for the connection to LDAP: -

```
./subsystem=elytron/dir-context=ldap-connection:add(url=ldap://localhost:10389,
principal="uid=admin,ou=system", credential-reference={clear-text=secret})
```

Then a security realm can be created to search LDAP and verify the supplied password: -

```
./subsystem=elytron/ldap-realm=ldap-realm:add(dir-context=ldap-connection, \
direct-verification=true, \
identity-mapping={search-base-dn="ou=users,dc=group-to-principal,dc=wildfly,dc=org", \
rdn-identifier="uid", \
attribute-mapping=[{filter-base-dn="ou=groups,dc=group-to-principal,dc=wildfly,dc=org",filter="(un
```

In the prior two examples information is loaded from LDAP to use directly as groups or roles, in the Elytron case information can be loaded from LDAP to associate with the identity as attributes - these can subsequently be mapped to roles but attributes can be loaded for other purposes as well.

By default, if no `role-decoder` is defined for given `security-domain`, identity attribute "`Roles`" is mapped to the identity roles.

This leads to the following configuration.

```
<subsystem xmlns="urn:wildfly:elytron:1.0" final-providers="combined-providers"
disallowed-providers="OracleUcrypto">
    ...
    <security-realms>
      ...
      <ldap-realm name="ldap-realm" dir-context="ldap-connection" direct-verification="true">
        <identity-mapping rdn-identifier="uid"
search-base-dn="ou=users,dc=group-to-principal,dc=wildfly,dc=org">
          <attribute-mapping>
            <attribute from="uid" to="Roles" filter="(uniqueMember={1})"
filter-base-dn="ou=groups,dc=group-to-principal,dc=wildfly,dc=org"/>
          </attribute-mapping>
        </identity-mapping>
      </ldap-realm>
    </security-realms>
    ...
    <dir-contexts>
      <dir-context name="ldap-connection" url="ldap://localhost:10389"
principal="uid=admin,ou=system">
        <credential-reference clear-text="secret"/>
      </dir-context>
    </dir-contexts>
  </subsystem>
```

Composite Stores Migration

When using either PicketBox or the legacy security realms it is possible to define a configuration where authentication is performed against one identity store whilst the information used for authorization is loaded from a different store, when using WildFly Elytron this can be achieved by using an aggregate security realm.

The example here makes use of a properties file for authentication and then searches LDAP to load group / role information. Both of these are based on the previous examples within this document so the environmental information is not repeated here.

PicketBox Based Configuration

A PicketBox based security domain can be created by using the following CLI commands: -

```
./subsystem=security/security-domain=application-security:add
./subsystem=security/security-domain=application-security/authentication=classic:add(login-modules=
\
{code=UsersRoles, flag=Required, module-options={ \
password-stacking=useFirstPass, \
usersProperties=file://${jboss.server.config.dir}/example-users.properties, \
rolesProperties=file://${jboss.server.config.dir}/example-roles.properties}} \
{code=LdapExtended, flag=Required, module-options={ \
password-stacking=useFirstPass, \
java.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory, \
java.naming.provider.url=ldap://localhost:10389, \
java.naming.security.authentication=simple, \
bindDN="uid=admin,ou=system", \
bindCredential=secret, \
baseCtxDN="ou=users,dc=group-to-principal,dc=wildfly,dc=org", \
baseFilter="(uid={0})", \
rolesCtxDN="ou=groups,dc=group-to-principal,dc=wildfly,dc=org",\
roleFilter="(uniqueMember={1})", \
roleAttributeID="uid" \
}}])
```

This results in the following domain definition: -

```
<security-domain name="application-security">
    <authentication>
      <login-module code="UsersRoles" flag="required">
        <module-option name="password-stacking" value="useFirstPass"/>
        <module-option name="usersProperties"
value="file://${jboss.server.config.dir}/example-users.properties"/>
        <module-option name="rolesProperties"
value="file://${jboss.server.config.dir}/example-roles.properties"/>
      </login-module>
      <login-module code="LdapExtended" flag="required">
        <module-option name="password-stacking" value="useFirstPass"/>
        <module-option name="java.naming.factory.initial"
value="com.sun.jndi.ldap.LdapCtxFactory"/>
        <module-option name="java.naming.provider.url" value="ldap://localhost:10389"/>
        <module-option name="java.naming.security.authentication" value="simple"/>
        <module-option name="bindDN" value="uid=admin,ou=system"/>
        <module-option name="bindCredential" value="secret"/>
        <module-option name="baseCtxDN"
value="ou=users,dc=group-to-principal,dc=wildfly,dc=org"/>
        <module-option name="baseFilter" value="(uid={0})"/>
        <module-option name="rolesCtxDN"
value="ou=groups,dc=group-to-principal,dc=wildfly,dc=org"/>
        <module-option name="roleFilter" value="(uniqueMember={1})"/>
        <module-option name="roleAttributeID" value="uid"/>
      </login-module>
    </authentication>
  </security-domain>
```

During an authentication attempt the 'UsersRoles' login module will first be called to perform authentication based on the supplied credential, then the 'LdapExtLoginModule' will be called which will proceed to query LDAP to load the roles for the identity.

Legacy Security Realm Configuration

An equivalent configuration can also be created using the legacy security realms with the following commands: -

```
./core-service=management/ldap-connection=MyLdapConnection:add(url="ldap://localhost:10389",
search-dn="uid=admin,ou=system", search-credential="secret")

./core-service=management/security-realm=ApplicationSecurity:add
./core-service=management/security-realm=ApplicationSecurity/authentication=properties:add(path=ex
relative-to=jboss.server.config.dir, plain-text=true)

batch
./core-service=management/security-realm=ApplicationSecurity/authorization=ldap:add(connection=MyL
base-dn="ou=users,dc=group-to-principal,dc=wildfly,dc=org")
./core-service=management/security-realm=ApplicationSecurity/authorization=ldap/group-search=group
iterative=true, prefer-original-connection=true, principal-attribute=uniqueMember,
search-by=DISTINGUISHED_NAME, group-name=SIMPLE, group-name-attribute=uid)
run-batch
```

This results in the following realm definition: -

```
<security-realm name="ApplicationSecurity">
    <authentication>
      <properties path="example-users.properties" relative-to="jboss.server.config.dir"
plain-text="true"/>
    </authentication>
    <authorization>
      <ldap connection="MyLdapConnection">
        <username-to-dn>
          <username-filter base-dn="ou=users,dc=group-to-principal,dc=wildfly,dc=org"
attribute="uid"/>
        </username-to-dn>
        <group-search group-name="SIMPLE" iterative="true" group-name-attribute="uid">
          <group-to-principal search-by="DISTINGUISHED_NAME"
base-dn="ou=groups,dc=group-to-principal,dc=wildfly,dc=org" prefer-original-connection="true">
            <membership-filter principal-attribute="uniqueMember"/>
          </group-to-principal>
        </group-search>
      </ldap>
    </authorization>
  </security-realm>

  <outbound-connections>
    <ldap name="MyLdapConnection" url="ldap://localhost:10389" search-dn="uid=admin,ou=system"
search-credential="secret"/>
  </outbound-connections>
```

As with the PicketBox example, authentication is first performed using the properties file - then group searching is performed against LDAP.

Migrated WildFly Elytron Configuration

The equivalent WildFly Elytron configuration can be defined with the following commands: -

```
./subsystem=elytron/dir-context=ldap-connection:add(url=ldap://localhost:10389,
principal="uid=admin,ou=system", credential-reference={clear-text=secret})

./subsystem=elytron/ldap-realm=ldap-realm:add(dir-context=ldap-connection, \
direct-verification=true, \
identity-mapping={search-base-dn="ou=users,dc=group-to-principal,dc=wildfly,dc=org", \
rdn-identifier="uid", \
attribute-mapping=[{filter-base-dn="ou=groups,dc=group-to-principal,dc=wildfly,dc=org",filter="(un
relative-to=jboss.server.config.dir, plain-text=true, digest-realm-name="Application Security"},
groups-properties={path=example-roles.properties, relative-to=jboss.server.config.dir},
groups-attribute=Roles)

./subsystem=elytron/aggregate-realm=combined-realm:add(authentication-realm=application-properties
authorization-realm=ldap-realm)

./subsystem=elytron/security-domain=application-security:add(realms=[{realm=combined-realm}],
default-realm=combined-realm, permission-mapper=default-permission-mapper)
./subsystem=elytron/http-authentication-factory=application-security-http:add(http-server-mechanis
security-domain=application-security, mechanism-configurations=[{mechanism-name=BASIC}])
```

This results in the following definitions: -

```
<subsystem xmlns="urn:wildfly:elytron:1.1" final-providers="combined-providers"
disallowed-providers="OracleUcrypto">
    ...
    <security-domains>
      ...
      <security-domain name="application-security" default-realm="combined-realm"
permission-mapper="default-permission-mapper">
        <realm name="combined-realm"/>
      </security-domain>
    </security-domains>
    <security-realms>
      <aggregate-realm name="combined-realm" authentication-realm="application-properties"
authorization-realm="ldap-realm"/>
        ...
        <properties-realm name="application-properties" groups-attribute="Roles">
          <users-properties path="example-users.properties"
relative-to="jboss.server.config.dir" digest-realm-name="Application Security"
plain-text="true"/>
          <groups-properties path="example-roles.properties"
relative-to="jboss.server.config.dir"/>
        </properties-realm>
        <ldap-realm name="ldap-realm" dir-context="ldap-connection" direct-verification="true">
          <identity-mapping rdn-identifier="uid"
search-base-dn="ou=users,dc=group-to-principal,dc=wildfly,dc=org">
            <attribute-mapping>
              <attribute from="uid" to="Roles" filter="(uniqueMember={1})"
filter-base-dn="ou=groups,dc=group-to-principal,dc=wildfly,dc=org"/>
            </attribute-mapping>
          </identity-mapping>
        </ldap-realm>
    </security-realms>
    ...
    <http>
      ...
      <http-authentication-factory name="application-security-http"
http-server-mechanism-factory="global" security-domain="application-security">
        <mechanism-configuration>
          <mechanism mechanism-name="BASIC"/>
        </mechanism-configuration>
      </http-authentication-factory>
      ...
    </http>
    ...
    <dir-contexts>
      <dir-context name="ldap-connection" url="ldap://localhost:10389"
principal="uid=admin,ou=system">
        <credential-reference clear-text="secret"/>
      </dir-context>
    </dir-contexts>
  </subsystem>
```

Within the WildFly Elytron example a new security realm 'aggregate-realm' has been defined, this definition specifies which of the defined security realms should be used for the authentication step and which of the security realms should be used for the loading of the identity used for subsequent authorization decisions.

Database Authentication

The section describing how to migrate from database accessible via JDBC datasource based authentication using PicketBox to Elytron. This section will illustrate some equivalent configuration using PicketBox security domains and show the equivalent configuration using Elytron but will not repeat the steps to wire it all together covered in the previous sections.

These configuration examples are developed against a test database with users table like:

```
CREATE TABLE User (
    id BIGINT NOT NULL,
    username VARCHAR(255),
    password VARCHAR(255),
    role ENUM('admin', 'manager', 'user'),
    PRIMARY KEY (id),
    UNIQUE (username)
)
```

For authentication purposes the username will be matched against the '`username`' column, password will be expected in hex-encoded MD5 hash in '`password`' column. User role for authorization purposes will be taken from '`role`' column.

PicketBox Database LoginModule

The following commands can create a PicketBox security domain configured to use database accessible via JDBC datasource to verify a username and password and to assign roles.

```
./subsystem=security/security-domain=application-security/:add
./subsystem=security/security-domain=application-security/authentication=classic:add(login-modules=
flag=Required, module-options={ \
    dsJndiName="java:jboss/datasources/ExampleDS", \
    principalsQuery="SELECT password FROM User WHERE username = ?", \
    rolesQuery="SELECT role, 'Roles' FROM User WHERE username = ?", \
    hashAlgorithm=MD5, \
    hashEncoding=base64 \
}})
```

This results in the following configuration.

```
<subsystem xmlns="urn:jboss:domain:security:2.0">
            <security-domains>
                ...
                <security-domain name="application-security">
                    <authentication>
                        <login-module code="Database" flag="required">
                            <module-option name="dsJndiName"
value="java:jboss/datasources/ExampleDS"/>
                            <module-option name="principalsQuery" value="SELECT password FROM
User WHERE username = ?"/>
                            <module-option name="rolesQuery" value="SELECT role, 'Roles' FROM
User WHERE username = ?"/>
                            <module-option name="hashAlgorithm" value="MD5"/>
                            <module-option name="hashEncoding" value="base64"/>
                        </login-module>
                    </authentication>
                </security-domain>
            </security-domains>
        </subsystem>
```

Migrated

Within the Elytron subsystem to use database accesible via JDBC you need to define `jdbc-realm`:

```
./subsystem=elytron/jdbc-realm=jdbc-realm:add(principal-query=[{ \
    data-source=ExampleDS, \
    sql="SELECT role, password FROM User WHERE username = ?", \
    attribute-mapping=[{index=1, to=Roles}] \
    simple-digest-mapper={algorithm=simple-digest-md5, password-index=2}, \
}])
```

This results in the following overall configuration:

```
<subsystem xmlns="urn:wildfly:elytron:1.0" final-providers="combined-providers"
disallowed-providers="OracleUcrypto">
            ...
            <security-realms>
                ...
                <jdbc-realm name="jdbc-realm">
                    <principal-query sql="SELECT role, password FROM User WHERE username = ?"
data-source="ExampleDS">
                        <attribute-mapping>
                            <attribute to="Roles" index="1"/>
                        </attribute-mapping>
                        <simple-digest-mapper password-index="2"/>
                    </principal-query>
                </jdbc-realm>
                ...
            </security-realms>
            ...
        </subsystem>
```

In comparison with PicketBox solution, Elytron `jdbc-realm` use one SQL query to obtain all user attributes and credentials. Their extraction from SQL result specifies mappers.

N-M relation beetween user and roles

When using a n:m-relation beetween user and roles (which means: the user has multiple roles), the previous configuration does not work.

The database:

```
CREATE TABLE User (
    id BIGINT NOT NULL,
    username VARCHAR(255),
    password VARCHAR(255),
    PRIMARY KEY (id),
    UNIQUE (username)
)

CREATE TABLE Role(
    id BIGINT NOT NULL,
    rolename VARCHAR(255),
    PRIMARY KEY (id),
    UNIQUE (rolename)
)

CREATE TABLE Userrole(
    userid BIGINT not null,
    roleid BIGINT not null,
    PRIMARY KEY (userid, roleid),
    FOREIGN KEY (userid) references User(id,
    FOREIGN KEY (roleid) references Role(id)
)
```

Here you need two configure two principal queries:

```
<jdbc-realm name="jdbc-realm">
  <principal-query sql="SELECT PASSWORD FROM USER WHERE USERNAME = ?" data-source="ExampleDS">
   <clear-password-mapper password-index="1"/>
  </principal-query>
  <principal-query sql="SELECT R.ROLENAME from ROLE AS R, USERROLE AS UR, USER AS U WHERE
U.USERNAME=? AND UR.ROLEID = R.ID AND UR.USERID = U.ID" data-source="ExampleDS">
   <attribute-mapping>
    <attribute to="roles" index="1"/>
   </attribute-mapping>
  </principal-query>
 </jdbc-realm>
```

The second query needs an attribute mapping to decode the selected rolename column (index 1):

```
<mappers>
  ...
  <simple-role-decoder name="from-roles-attribute" attribute="roles"/>
  ...
 </mappers>
```

The role decoder is referenced by the security domain:

```
<security-domain name="MyDomain" default-realm="jdbc-realm"
permission-mapper="default-permission-mapper">
  <realm name="MyDbRealm" role-decoder="from-roles-attribute"/>
 </security-domain>
```

Kerberos Authentication Migration

When working with Kerberos configuration it is possible for the application server to rely on configuration from the environment or the key configuration can be specified using system properties, for the purpose of these examples I define system properties - these properties are applicable to both the legacy configuration and the migrated Elytron configuration.

```
./system-property=sun.security.krb5.debug:add(value=true)
./system-property=java.security.krb5.realm:add(value=ELYTRON.ORG)
./system-property=java.security.krb5.kdc:add(value=kdc.elytron.org)
```

The first line makes debugging easier but the last two lines specify the Kerberos realm in use and the address of the KDC.

```
<system-properties>
    <property name="sun.security.krb5.debug" value="true"/>
    <property name="java.security.krb5.realm" value="ELYTRON.ORG"/>
    <property name="java.security.krb5.kdc" value="kdc.elytron.org"/>
  </system-properties>
```

HTTP Authentication

Legacy Security Realm

A legacy security realm can be define so that SPNEGO authentication can be enabled for the HTTP management interface.

```
./core-service=management/security-realm=Kerberos:add
./core-service=management/security-realm=Kerberos/server-identity=kerberos:add
./core-service=management/security-realm=Kerberos/server-identity=kerberos/keytab=HTTP\/test-serve
debug=true)
./core-service=management/security-realm=Kerberos/authentication=kerberos:add(remove-realm=true)
```

This results in the following configuration: -

```
<security-realms>
    ...
    <security-realm name="Kerberos">
      <server-identities>
        <kerberos>
          <keytab principal="HTTP/test-server.elytron.org@ELYTRON.ORG"
path="/home/darranl/src/kerberos/test-server.keytab" debug="true"/>
        </kerberos>
      </server-identities>
      <authentication>
        <kerberos remove-realm="true"/>
      </authentication>
    </security-realm>
  </security-realms>
```

Application SPNEGO

Alternatively deployed applications would make use of a pair of security domains.

```
./subsystem=security/security-domain=host:add
./subsystem=security/security-domain=host/authentication=classic:add
./subsystem=security/security-domain=host/authentication=classic/login-module=1:add(code=Kerberos,
flag=Required, module-options={storeKey=true, useKeyTab=true,
principal=HTTP/test-server.elytron.org@ELYTRON.ORG,
keyTab=/home/darranl/src/kerberos/test-server.keytab, debug=true}
```

```
./subsystem=security/security-domain=SPNEGO:add
./subsystem=security/security-domain=SPNEGO/authentication=classic:add
./subsystem=security/security-domain=SPNEGO/authentication=classic/login-module=1:add(code=SPNEGO,
flag=requisite,  module-options={password-stacking=useFirstPass, serverSecurityDomain=host})
./subsystem=security/security-domain=SPNEGO/authentication=classic/login-module=1:write-attribute(
value=org.jboss.security.negotiation)
./subsystem=security/security-domain=SPNEGO/authentication=classic/login-module=2:add(code=UsersRo
flag=required, module-options={password-stacking=useFirstPass,
usersProperties=file:///home/darranl/src/kerberos/spnego-users.properties,
rolesProperties=file:///home/darranl/src/kerberos/spnego-roles.properties,
defaultUsersProperties=file:///home/darranl/src/kerberos/spnego-users.properties,
defaultRolesProperties=file:///home/darranl/src/kerberos/spnego-roles.properties})
```

This results in: -

```
<subsystem xmlns="urn:jboss:domain:security:2.0">
    <security-domains>
      ...
      <security-domain name="host">
        <authentication>
          <login-module name="1" code="Kerberos" flag="required">
            <module-option name="storeKey" value="true"/>
            <module-option name="useKeyTab" value="true"/>
            <module-option name="principal" value="HTTP/test-server.elytron.org@ELYTRON.ORG"/>
            <module-option name="keyTab" value="/home/darranl/src/kerberos/test-server.keytab"/>
            <module-option name="debug" value="true"/>
          </login-module>
        </authentication>
      </security-domain>
      <security-domain name="SPNEGO">
        <authentication>
          <login-module name="1" code="SPNEGO" flag="requisite"
module="org.jboss.security.negotiation">
            <module-option name="password-stacking" value="useFirstPass"/>
            <module-option name="serverSecurityDomain" value="host"/>
          </login-module>
          <login-module name="2" code="UsersRoles" flag="required">
            <module-option name="password-stacking" value="useFirstPass"/>
            <module-option name="usersProperties"
value="file:///home/darranl/src/kerberos/spnego-users.properties"/>
            <module-option name="rolesProperties"
value="file:///home/darranl/src/kerberos/spnego-roles.properties"/>
            <module-option name="defaultUsersProperties"
value="file:///home/darranl/src/kerberos/spnego-users.properties"/>
            <module-option name="defaultRolesProperties"
value="file:///home/darranl/src/kerberos/spnego-roles.properties"/>
          </login-module>
        </authentication>
      </security-domain>
    </security-domains>
  </subsystem>
```

An application can now be deployed referencing the SPNEGO security domain and secured with SPNEGO mechanism.

Migrated SPNEGO

The equivalent configuration can be achieved with WildFly Elytron by first defining a security realm which will be used to load identity information.

```
./subsystem=elytron/properties-realm=spnego-properties:add(users-properties={path=/home/darranl/sr
plain-text=true, digest-realm-name=ELYTRON.ORG},
groups-properties={path=/home/darranl/src/kerberos/spnego-roles.properties})
```

Next a Kerberos security factory is defined which allows the server to load it's own Kerberos identity.

```
./subsystem=elytron/kerberos-security-factory=test-server:add(path=/home/darranl/src/kerberos/test
principal=HTTP/test-server.elytron.org@ELYTRON.ORG, debug=true)
```

As with the previous examples we define a security realm to pull together the policy as well as a HTTP authentication factory for the authentication policy.

```
./subsystem=elytron/security-domain=SPNEGODomain:add(default-realm=spnego-properties,
realms=[{realm=spnego-properties, role-decoder=groups-to-roles}],
permission-mapper=default-permission-mapper)
./subsystem=elytron/http-authentication-factory=spnego-http-authentication:add(security-domain=SPN
http-server-mechanism-factory=global,mechanism-configurations=[{mechanism-name=SPNEGO,
credential-security-factory=test-server}])
```

Overall this results in the following configuration: -

```
<subsystem xmlns="urn:wildfly:elytron:1.0" final-providers="combined-providers"
disallowed-providers="OracleUcrypto">
    ...
    <security-domains>
    ...
      <security-domain name="SPNEGODomain" default-realm="spnego-properties"
permission-mapper="default-permission-mapper">
        <realm name="spnego-properties" role-decoder="groups-to-roles"/>
      </security-domain>
    </security-domains>
    <security-realms>
      ...
      <properties-realm name="spnego-properties">
        <users-properties path="/home/darranl/src/kerberos/spnego-users.properties"
digest-realm-name="ELYTRON.ORG" plain-text="true"/>
        <groups-properties path="/home/darranl/src/kerberos/spnego-roles.properties"/>
      </properties-realm>
    </security-realms>
    <credential-security-factories>
      <kerberos-security-factory name="test-server"
principal="HTTP/test-server.elytron.org@ELYTRON.ORG"
path="/home/darranl/src/kerberos/test-server.keytab" debug="true"/>
    </credential-security-factories>
    ...
    <http>
      ...
      <http-authentication-factory name="spnego-http-authentication"
http-server-mechanism-factory="global" security-domain="SPNEGODomain">
        <mechanism-configuration>
          <mechanism mechanism-name="SPNEGO" credential-security-factory="test-server"/>
        </mechanism-configuration>
      </http-authentication-factory>
      ...
    </http>
    ...
  </subsystem>
```

Now, to enable SPNEGO authentication for the HTTP management interface, update this interface to reference the `http-authentication-factory` defined above, as described in the properties authentication section.

Alternatively, to secure an application using SPNEGO authentication, an application security domain can be defined in the Undertow subsystem to map security domains to the `http-authentication-factory` defined above, as described in the properties authentication section.

Remoting / SASL Authentication

Legacy Security Realm

It is also possible to define a legacy security realm for Kerberos / GSSAPI SASL authenticatio for Remoting authentication such as the native management interface.

```
./core-service=management/security-realm=Kerberos:add
./core-service=management/security-realm=Kerberos/server-identity=kerberos:add
./core-service=management/security-realm=Kerberos/server-identity=kerberos/keytab=remote\/test-ser
debug=true)
./core-service=management/security-realm=Kerberos/authentication=kerberos:add(remove-realm=true)
```

```
<management>
    <security-realms>
      ...
      <security-realm name="Kerberos">
        <server-identities>
          <kerberos>
            <keytab principal="remote/test-server.elytron.org@ELYTRON.ORG"
path="/home/darranl/src/kerberos/remote-test-server.keytab" debug="true"/>
          </kerberos>
        </server-identities>
        <authentication>
          <kerberos remove-realm="true"/>
        </authentication>
      </security-realm>
    </security-realms>
    ...
  </management>
```

Migrated GSSAPI

The steps to define the equivalent Elytron configuration are very similar to the HTTP example.

First define the security realm to load the identity from: -

```
./path=kerberos:add(relative-to=user.home, path=src/kerberos)
./subsystem=elytron/properties-realm=kerberos-properties:add(users-properties={path=kerberos-users
relative-to=kerberos, digest-realm-name=ELYTRON.ORG},
groups-properties={path=kerberos-groups.properties, relative-to=kerberos})
```

Then define the Kerberos security factory for the server's identity.

```
./subsystem=elytron/kerberos-security-factory=test-server:add(relative-to=kerberos,
path=remote-test-server.keytab, principal=remote/test-server.elytron.org@ELYTRON.ORG)
```

Finally define the security domain and this time a SASL authentication factory.

```
./subsystem=elytron/security-domain=KerberosDomain:add(default-realm=kerberos-properties,
realms=[{realm=kerberos-properties, role-decoder=groups-to-roles}],
permission-mapper=default-permission-mapper)
./subsystem=elytron/sasl-authentication-factory=gssapi-authentication-factory:add(security-domain=
sasl-server-factory=elytron, mechanism-configurations=[{mechanism-name=GSSAPI,
credential-security-factory=test-server}])
```

This results in the following subsystem configuration: -

```
<subsystem xmlns="urn:wildfly:elytron:1.0" final-providers="combined-providers"
disallowed-providers="OracleUcrypto">
    ...
    <security-domains>
      ...
      <security-domain name="KerberosDomain" default-realm="kerberos-properties"
permission-mapper="default-permission-mapper">
        <realm name="kerberos-properties" role-decoder="groups-to-roles"/>
      </security-domain>
    </security-domains>
    <security-realms>
     ...
       <properties-realm name="kerberos-properties">
         <users-properties path="kerberos-users.properties" relative-to="kerberos"
digest-realm-name="ELYTRON.ORG"/>
         <groups-properties path="kerberos-groups.properties" relative-to="kerberos"/>
       </properties-realm>
     </security-realms>
     <credential-security-factories>
       <kerberos-security-factory name="test-server"
principal="remote/test-server.elytron.org@ELYTRON.ORG" path="remote-test-server.keytab"
relative-to="kerberos"/>
     </credential-security-factories>
     ...
     <sasl>
       ...
       <sasl-authentication-factory name="gssapi-authentication-factory"
sasl-server-factory="elytron" security-domain="KerberosDomain">
         <mechanism-configuration>
           <mechanism mechanism-name="GSSAPI" credential-security-factory="test-server"/>
         </mechanism-configuration>
       </sasl-authentication-factory>
       ...
     </sasl>
   </subsystem>
```

The management interface or Remoting connectors can now be updated to reference the SASL authentication factory.

The two Elytron examples defined here could also be combined into one to use a shared security domain and security realm and just use protocol specific authentication factories each referencing their own Kerberos security factory.

Caching Migration

Where a PicketBox based security domain is defined it is possible to enable caching for that security domain, this enables subsequent hits to the identity store to be avoided as an in memory cache can be used instead, this example demonstrates how caching can be used with a WildFly Elytron based configuration.

The purpose of this chapter is to highlight the migration of a configuration with caching enabled, this example is based in the previous LDAP example but with caching enabled.

PicketBox Example

A PicketBox based security domain can be defined with the following commands.

```
./subsystem=security/security-domain=application-security:add(cache-type=default)
./subsystem=security/security-domain=application-security/authentication=classic:add(login-modules=
flag=Required, module-options={ \
java.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory, \
java.naming.provider.url=ldap://localhost:10389, \
java.naming.security.authentication=simple, \
bindDN="uid=admin,ou=system", \
bindCredential=secret, \
baseCtxDN="ou=users,dc=group-to-principal,dc=wildfly,dc=org", \
baseFilter="(uid={0})", \
rolesCtxDN="ou=groups,dc=group-to-principal,dc=wildfly,dc=org",\
roleFilter="(uniqueMember={1})", \
roleAttributeID="uid" \
}])
```

Resulting in the following security domain definition: -

```
<subsystem xmlns="urn:jboss:domain:security:2.0">
    <security-domains>
      ...
      <security-domain name="application-security" cache-type="default">
        <authentication>
          <login-module code="LdapExtended" flag="required">
            <module-option name="java.naming.factory.initial"
value="com.sun.jndi.ldap.LdapCtxFactory"/>
            <module-option name="java.naming.provider.url" value="ldap://localhost:10389"/>
            <module-option name="java.naming.security.authentication" value="simple"/>
            <module-option name="bindDN" value="uid=admin,ou=system"/>
            <module-option name="bindCredential" value="secret"/>
            <module-option name="baseCtxDN"
value="ou=users,dc=group-to-principal,dc=wildfly,dc=org"/>
            <module-option name="baseFilter" value="(uid={0})"/>
            <module-option name="rolesCtxDN"
value="ou=groups,dc=group-to-principal,dc=wildfly,dc=org"/>
            <module-option name="roleFilter" value="(uniqueMember={1})"/>
            <module-option name="roleAttributeID" value="uid"/>
          </login-module>
        </authentication>
      </security-domain>
    </security-domains>
  </subsystem>
```

The key difference to the raw LDAP example is that a cache-type of 'default' has been specified on the security domain. The default cache-type is an in memory cache, when using PicketBox it is also possible to specify a cache-type of 'infinispan' although this is not supported with WildFly Elytron as various aspects of a SecurityIdentity are not suitable for replication.

Migrated Example

When using WildFly Elytron where caching is required the individual security realm is wrapped using a cache, a migrated configuration can be defined with the following commands: -

```
./subsystem=elytron/dir-context=ldap-connection:add(url=ldap://localhost:10389,
principal="uid=admin,ou=system", credential-reference={clear-text=secret})
./subsystem=elytron/ldap-realm=ldap-realm:add(dir-context=ldap-connection, \
direct-verification=true, \
identity-mapping={search-base-dn="ou=users,dc=group-to-principal,dc=wildfly,dc=org", \
rdn-identifier="uid", \
attribute-mapping=[{filter-base-dn="ou=groups,dc=group-to-principal,dc=wildfly,dc=org",filter="(un
```

These can then be used in a security domain and subsequently an authentication factory.

```
./subsystem=elytron/security-domain=application-security:add(realms=[{realm=cached-ldap}],
default-realm=cached-ldap, permission-mapper=default-permission-mapper)
./subsystem=elytron/http-authentication-factory=application-security-http:add(http-server-mechanis
security-domain=application-security, mechanism-configurations=[{mechanism-name=BASIC}])
```

In this final step it is very important that the caching-realm is referenced rather than the original realm otherwise caching will be bypassed.

This results in the following definitions: -

```
<subsystem xmlns="urn:wildfly:elytron:1.1" final-providers="combined-providers"
disallowed-providers="OracleUcrypto">
    ...
    <security-domains>
      ...
      <security-domain name="application-security" default-realm="cached-ldap"
permission-mapper="default-permission-mapper">
        <realm name="cached-ldap"/>
      </security-domain>
    </security-domains>
    <security-realms>
      ...
      <ldap-realm name="ldap-realm" dir-context="ldap-connection" direct-verification="true">
        <identity-mapping rdn-identifier="uid"
search-base-dn="ou=users,dc=group-to-principal,dc=wildfly,dc=org">
          <attribute-mapping>
            <attribute from="uid" to="Roles" filter="(uniqueMember={1})"
filter-base-dn="ou=groups,dc=group-to-principal,dc=wildfly,dc=org"/>
          </attribute-mapping>
        </identity-mapping>
      </ldap-realm>
      <caching-realm name="cached-ldap" realm="ldap-realm"/>
    </security-realms>
    ...
    <http>
      ...
      <http-authentication-factory name="application-security-http"
http-server-mechanism-factory="global" security-domain="application-security">
        <mechanism-configuration>
          <mechanism mechanism-name="BASIC"/>
        </mechanism-configuration>
      </http-authentication-factory>
      ...
    </http>
    ...
    <dir-contexts>
      <dir-context name="ldap-connection" url="ldap://localhost:10389"
principal="uid=admin,ou=system">
        <credential-reference clear-text="secret"/>
      </dir-context>
    </dir-contexts>
  </subsystem>
```

15.11.2 Clients

Application Client Migration

Naming Client

This migration example assumes a client application performs a remote JNDI lookup using an `InitialContext` backed by the `org.jboss.naming.remote.client.InitialContextFactory` class.

Original Configuration

An `InitialContext` backed by the `org.jboss.naming.remote.client.InitialContextFactory` class can be created by specifying properties that contain the URL of the naming provider to connect to along with appropriate user credentials:

```
Properties properties = new Properties();
properties.put(Context.INITIAL_CONTEXT_FACTORY,
"org.jboss.naming.remote.client.InitialContextFactory");
properties.put(Context.PROVIDER_URL, "http-remoting://127.0.0.1:8080");
properties.put(Context.SECURITY_PRINCIPAL, "bob");
properties.put(Context.SECURITY_CREDENTIALS, "secret");
InitialContext context = new InitialContext(properties);
Bar bar = (Bar) context.lookup("foo/bar");
...
```

Migrated Configuration

An `InitialContext` backed by the `org.wildfly.naming.client.WildFlyInitialContextFactory` class can be created by specifying a property that contains the URL of the naming provider to connect to. The user credentials can be specified using a WildFly client configuration file or programmatically.

Configuration File Approach

A `wildfly-config.xml` file that contains the user credentials to use when establishing a connection to the naming provider can be added to the client application's `META-INF` directory:

wildfly-config.xml

```
<configuration>
    <authentication-client xmlns="urn:elytron:1.0">
        <authentication-rules>
            <rule use-configuration="namingConfig">
                <match-host name="127.0.0.1"/>
            </rule>
        </authentication-rules>
        <authentication-configurations>
            <configuration name="namingConfig">
                <set-user-name name="bob"/>
                <credentials>
                    <clear-password password="secret"/>
                </credentials>
            </configuration>
        </authentication-configurations>
    </authentication-client>
</configuration>
```

An `InitialContext` can then be created as follows:

```
Properties properties = new Properties();
properties.put(Context.INITIAL_CONTEXT_FACTORY,
"org.wildfly.naming.client.WildFlyInitialContextFactory");
properties.put(Context.PROVIDER_URL, "remote+http://127.0.0.1:8080");
InitialContext context = new InitialContext(properties);
Bar bar = (Bar) context.lookup("foo/bar");
...
```

Programmatic Approach

The user credentials to use when establishing a connection to the naming provider can be specified directly in the client application's code:

```
// create your authentication configuration
AuthenticationConfiguration namingConfig =
AuthenticationConfiguration.empty().useName("bob").usePassword("secret");

// create your authentication context
AuthenticationContext context =
AuthenticationContext.empty().with(MatchRule.ALL.matchHost("127.0.0.1"), namingConfig);

// create a callable that creates and uses an InitialContext
Callable<Void> callable = () -> {
    Properties properties = new Properties();
    properties.put(Context.INITIAL_CONTEXT_FACTORY,
"org.wildfly.naming.client.WildFlyInitialContextFactory");
    properties.put(Context.PROVIDER_URL, "remote+http://127.0.0.1:8080");
    InitialContext context = new InitialContext(properties);
    Bar bar = (Bar) context.lookup("foo/bar");
    ...
    return null;
};

// use your authentication context to run your callable
context.runCallable(callable);
```

EJB Client

This migration example assumes a client application is configured to invoke an EJB deployed on a remote server using a `jboss-ejb-client.properties` file.

Original Configuration

A `jboss-ejb-client.properties` file that contains the information needed to connect to the remote server can be specified in a client application's `META-INF` directory:

jboss-ejb-client.properties

```
remote.connectionprovider.create.options.org.xnio.Options.SSL_ENABLED=false
remote.connections=default
remote.connection.default.host=127.0.0.1
remote.connection.default.port = 8080
remote.connection.default.username=bob
remote.connection.default.password=secret
```

An EJB can then be looked up and a method can be invoked on it as follows:

```
// create an InitialContext
Properties properties = new Properties();
properties.put(Context.INITIAL_CONTEXT_FACTORY,
"org.jboss.naming.remote.client.InitialContextFactory");
properties.put(Context.URL_PKG_PREFIXES, "org.jboss.ejb.client.naming");
InitialContext context = new InitialContext(properties);

// look up an EJB and invoke one of its methods
RemoteCalculator statelessRemoteCalculator = (RemoteCalculator) context.lookup(
    "ejb:/ejb-remote-server-side//CalculatorBean!" + RemoteCalculator.class.getName());
int sum = statelessRemoteCalculator.add(101, 202);
```

Migrated Configuration

The information needed to connect to the remote server can be specified using a WildFly client configuration file or programmatically.

Configuration File Approach

A `wildfly-config.xml` file that contains the information needed to connect to the remote server can be added to the client application's `META-INF` directory:

wildfly-config.xml

```
<configuration>
    <authentication-client xmlns="urn:elytron:1.0">
        <authentication-rules>
            <rule use-configuration="ejbConfig">
                <match-host name="127.0.0.1"/>
            </rule>
        </authentication-rules>
        <authentication-configurations>
            <configuration name="ejbConfig">
                <set-user-name name="bob"/>
                <credentials>
                    <clear-password password="secret"/>
                </credentials>
            </configuration>
        </authentication-configurations>
    </authentication-client>
    <jboss-ejb-client xmlns="urn:jboss:wildfly-client-ejb:3.0">
        <connections>
            <connection uri="remote+http://127.0.0.1:8080" />
        </connections>
    </jboss-ejb-client>
</configuration>
```

An EJB can then be looked up and a method can be invoked on it as follows:

```
// create an InitialContext
Properties properties = new Properties();
properties.put(Context.INITIAL_CONTEXT_FACTORY,
"org.wildfly.naming.client.WildFlyInitialContextFactory");
InitialContext context = new InitialContext(properties);

// look up an EJB and invoke one of its methods (same as before)
RemoteCalculator statelessRemoteCalculator = (RemoteCalculator) context.lookup(
    "ejb:/ejb-remote-server-side//CalculatorBean!" + RemoteCalculator.class.getName());
int sum = statelessRemoteCalculator.add(101, 202);
```

Programmatic Approach
The information needed to connect to the remote server can be specified directly in the client application's code:

```
// create your authentication configuration
AuthenticationConfiguration ejbConfig =
AuthenticationConfiguration.empty().useName("bob").usePassword("secret");

// create your authentication context
AuthenticationContext context =
AuthenticationContext.empty().with(MatchRule.ALL.matchHost("127.0.0.1"), ejbConfig);

// create a callable that invokes an EJB
Callable<Void> callable = () -> {

    // create an InitialContext
    Properties properties = new Properties();
    properties.put(Context.INITIAL_CONTEXT_FACTORY,
"org.wildfly.naming.client.WildFlyInitialContextFactory");
    properties.put(Context.PROVIDER_URL, "remote+http://127.0.0.1:8080");
    InitialContext context = new InitialContext(properties);

    // look up an EJB and invoke one of its methods (same as before)
    RemoteCalculator statelessRemoteCalculator = (RemoteCalculator) context.lookup(
        "ejb:/ejb-remote-server-side//CalculatorBean!" + RemoteCalculator.class.getName());
    int sum = statelessRemoteCalculator.add(101, 202);
    ...
    return null;
};

// use your authentication context to run your callable
context.runCallable(callable);
```

15.11.3 General Utilities

Security Vault Migration

Security Vault is primarily used in legacy configurations, a vault is used to store sensitive strings outside of the configuration files. WildFly server may only contain a single security vault.

Credential Store introduced in WildFly 11 is meant to expand Security Vault in terms of storing different credential types and introduce easy to implement SPI which allows to deploy custom implementations of CredentialStore SPI. Credentials are stored safely encrypted in storage file outside WildFly configuration files. Each WildFly server may contain multiple credential stores.

To easily migrate vault content into credential store we have added "vault" command into WildFly Elytron Tool. The tool could be found at $JBOSS_HOME/bin directory. It has several scripts named "elytron-tool.*" dependent on your platform of choice. One can use also simple form "java -jar $JBOSS_HOME/bin/wildfly-elytron-tool.jar <command> <arguments>" if it better suites ones needs.

Single Security Vault Conversion

To convert **single** security vault credential store use following example:

- to get sample vault use testing resources of Elytron Tool project from GitHub [1]

Command to run actual conversion:

```
./bin/elytron-tool.sh vault --enc-dir vault_data/ --keystore
vault-jceks.keystore --keystore-password MASK-2hKo56F1a3jYGnJwhPmiF5
--iteration 34 --salt 12345678 --alias test --location cs-v1.store --summary
```

Output:

```
Vault (enc-dir="vault_data/";keystore="vault-jceks.keystore") converted to
credential store "cs-v1.store"
Vault Conversion summary:
-------------------------------------
Vault Conversion Successful
CLI command to add new credential store:
/subsystem=elytron/credential-store=test:add(relative-to=jboss.server.data.dir,
```

Use elytron-tool.sh vault --help to get description of all parameters.

Notes:

- Elytron Tool cannot handle very first version of Security Vault data file.
- --keystore-password can come in two forms (1) masked as shown in the example or (2) clear text. Parameter --salt and --iteration are there to supply information to decrypt the masked password or to generate masked password in output. In case --salt and --iteration are omitted default values are used.
- When --summary parameter is specified, one can see nice output with CLI command to be used in WildFly console to add converted credential store to the configuration.

Bulk Security Vault Conversion

There is possibility to convert multiple vaults to credential store using --bulk-convert parameter with description file.
Example of description file from our tests [2]:

```
# Bulk conversion descriptor
keystore:target/test-classes/vault-v1/vault-jceks.keystore
keystore-password:MASK-2hKo56F1a3jYGnJwhPmiF5
enc-dir:target/test-classes/vault-v1/vault_data/
salt:12345678
iteration:34
location:target/v1-cs-1.store
alias:test

keystore:target/test-classes/vault-v1/vault-jceks.keystore
keystore-password:secretsecret
enc-dir:target/test-classes/vault-v1/vault_data/
location:target/v1-cs-2.store
alias:test

# different vault vault-v1-more
keystore:target/test-classes/vault-v1-more/vault-jceks.keystore
keystore-password:MASK-2hKo56F1a3jYGnJwhPmiF5
enc-dir:target/test-classes/vault-v1-more/vault_data/
salt:12345678
iteration:34
location:target/v1-cs-more.store
alias:test
```

After each "keystore:" option new conversion starts. All options are mandatory except "salt:", "iteration:" and "properties:"

Execute following command:

```
./bin/elytron-tool.sh vault --bulk-convert bulk-vault-conversion-desc
--summary
```

Output:

```
Vault
(enc-dir="vault-v1/vault_data/";keystore="vault-v1/vault-jceks.keystore")
converted to credential store "v1-cs-1.store"
Vault Conversion summary:
-------------------------------------
Vault Conversion Successful
CLI command to add new credential store:
/subsystem=elytron/credential-store=test:add(relative-to=jboss.server.data.dir,

-------------------------------------

Vault
(enc-dir="vault-v1/vault_data/";keystore="vault-v1/vault-jceks.keystore")
converted to credential store "v1-cs-2.store"
Vault Conversion summary:
-------------------------------------
Vault Conversion Successful
CLI command to add new credential store:
/subsystem=elytron/credential-store=test:add(relative-to=jboss.server.data.dir,

-------------------------------------

Vault
(enc-dir="vault-v1-more/vault_data/";keystore="vault-v1-more/vault-jceks.keysto
converted to credential store "v1-cs-more.store"
Vault Conversion summary:
-------------------------------------
Vault Conversion Successful
CLI command to add new credential store:
/subsystem=elytron/credential-store=test:add(relative-to=jboss.server.data.dir,

-------------------------------------
```

The result is conversion of all vaults with proper CLI commands.

References:

[1] https://github.com/wildfly-security/wildfly-elytron-tool/tree/master/src/test/resources/vault-v1
[2] https://github.com/wildfly-security/wildfly-elytron-tool/blob/master/src/test/java/org/wildfly/security/tool/VaultCo

Security Properties

Lets suppose security properties "a" and "c" defined in legacy security:

```
<subsystem xmlns="urn:jboss:domain:security:2.0">
            ...
            <security-properties>
                <property name="a" value="b" />
                <property name="c" value="d" />
            </security-properties>
        </subsystem>
```

To define security properties in Elytron subsystem you need to set attribute `security-properties` of the subsystem:

```
./subsystem=elytron:write-attribute(name=security-properties, value={ \
    a = "b", \
    c = "d" \
})
```

You can also add or change one another property without modification of others using map operations. Following command will set property "e":

```
./subsystem=elytron:map-put(name=security-properties, key=e, value=f)
```

By the same way you can also remove one of properties - in example newly created property "e":

```
./subsystem=elytron:map-remove(name=security-properties, key=e)
```

Output XML configuration will be:

```
<subsystem xmlns="urn:wildfly:elytron:1.0" final-providers="combined-providers"
disallowed-providers="OracleUcrypto">
            <security-properties>
                <security-property name="a" value="b"/>
                <security-property name="c" value="d"/>
            </security-properties>
            ...
        </subsystem>
```

15.11.4 SSL Migration

Simple SSL Migration

This section describe securing HTTP connections to the server using SSL using Elytron.
It suppose you have already configured SSL using legacy `security-realm`, for example by Admin Guide#Enable SSL, and your configuration looks like:

```
<security-realm name="ApplicationRealm">
  <server-identities>
    <ssl>
      <keystore path="server.keystore" relative-to="jboss.server.config.dir"
keystore-password="keystore_password" alias="server" key-password="key_password" />
    </ssl>
  </server-identities>
</security-realm>
```

To switch to Elytron you need to:

1. Create Elytron `key-store` - specifying where is the keystore file stored and password by which it is encrypted. Default type of keystore generated using keytool is JKS:

   ```
   /subsystem=elytron/key-store=LocalhostKeyStore:add(path=server.keystore,relative-to=jboss.se
   ```

2. Create Elytron `key-manager` - specifying keystore, alias (using `alias-filter`) and password of key:

   ```
   /subsystem=elytron/key-manager=LocalhostKeyManager:add(key-store=LocalhostKeyStore,alias-fil
   ```

3. Create Elytron `server-ssl-context` - specifying only reference to `key-manager` defined above:

   ```
   /subsystem=elytron/server-ssl-context=LocalhostSslContext:add(key-manager=LocalhostKeyManage
   ```

4. Switch `https-listener` from legacy `security-realm` to newly created Elytron `ssl-context`:

   ```
   /subsystem=undertow/server=default-server/https-listener=https:undefine-attribute(name=secur
   ```

5. And reload the server:

   ```
   reload
   ```

Output XML configuration of Elytron subsystem should look like:

```
<subsystem xmlns="urn:wildfly:elytron:1.0" ...>
            ...
            <tls>
                <key-stores>
                    <key-store name="LocalhostKeyStore">
                        <credential-reference clear-text="keystore_password"/>
                        <implementation type="JKS"/>
                        <file path="server.keystore" relative-to="jboss.server.config.dir"/>
                    </key-store>
                </key-stores>
                <key-managers>
                    <key-manager name="LocalhostKeyManager" key-store="LocalhostKeyStore">
                        <credential-reference clear-text="key_password"/>
                    </key-manager>
                </key-managers>
                <server-ssl-contexts>
                    <server-ssl-context name="LocalhostSslContext"
key-manager="LocalhostKeyManager"/>
                </server-ssl-contexts>
            </tls>
        </subsystem>
```

Output `https-listener` in Undertow subsystem should be:

```
<https-listener name="https" socket-binding="https" ssl-context="LocalhostSslContext"
enable-http2="true"/>
```

Client-Cert SSL Authentication Migration

This suppose you have already configured Client-Cert SSL authentication using `truststore` in legacy `security-realm`, for example by Admin Guide#Add Client-Cert to SSL, and your configuration looks like:

```
<security-realm name="ApplicationRealm">
  <server-identities>
    <ssl>
      <keystore path="server.keystore" relative-to="jboss.server.config.dir"
keystore-password="keystore_password" alias="server" key-password="key_password" />
    </ssl>
  </server-identities>
  <authentication>
    <truststore path="server.truststore" relative-to="jboss.server.config.dir"
keystore-password="truststore_password" />
    <local default-user="$local"/>
    <properties path="application-users.properties" relative-to="jboss.server.config.dir"/>
  </authentication>
</security-realm>
```

Following configuration is sufficient to prevent users without valid certificate and private key to access the server, but it does not provide user identity to the application. That require to define `CLIENT_CERT` HTTP mechanism / `EXTERNAL` SASL mechanism, which will be described later.)

At first use steps above to migrate basic part of the configuration. Then continue by following:

1. Create `key-store` of truststore - like for keystore above:

```
/subsystem=elytron/key-store=TrustStore:add(path=server.truststore,relative-to=jboss.server.
```

2. Create `trust-manager` - specifying `key-store` of trustore, created above:

```
/subsystem=elytron/trust-manager=TrustManager:add(key-store=TrustStore)
```

3. Modify `server-ssl-context` to use newly created trustmanager:

```
/subsystem=elytron/server-ssl-context=LocalhostSslContext:write-attribute(name=trust-manager
```

4. Enable client authentication for `server-ssl-context`:

```
/subsystem=elytron/server-ssl-context=LocalhostSslContext:write-attribute(name=need-client-a
```

5. And reload the server:

```
reload
```

Output XML configuration of Elytron subsystem should look like:

```
<subsystem xmlns="urn:wildfly:elytron:1.0" ...>
            ...
            <tls>
                <key-stores>
                    <key-store name="LocalhostKeyStore">
                        <credential-reference clear-text="keystore_password"/>
                        <implementation type="JKS"/>
                        <file path="server.keystore" relative-to="jboss.server.config.dir"/>
                    </key-store>
                    <key-store name="TrustStore">
                        <credential-reference clear-text="truststore_password"/>
                        <implementation type="JKS"/>
                        <file path="server.truststore" relative-to="jboss.server.config.dir"/>
                    </key-store>
                </key-stores>
                <key-managers>
                    <key-manager name="LocalhostKeyManager" key-store="LocalhostKeyStore"
alias-filter="server">
                        <credential-reference clear-text="key_password"/>
                    </key-manager>
                </key-managers>
                <trust-managers>
                    <trust-manager name="TrustManager" key-store="TrustStore"/>
                </trust-managers>
                <server-ssl-contexts>
                    <server-ssl-context name="LocalhostSslContext" need-client-auth="true"
key-manager="LocalhostKeyManager" trust-manager="TrustManager"/>
                </server-ssl-contexts>
            </tls>
        </subsystem>
```

SSL with Client Cert Migration

As this documentation is primarily intended for users migrating to WildFly Elytron I am going to jump straight into the configuration required with WildFly Elytron.

This section will cover how to create the various resources required to achieve CLIENT_CERT authentication with fallback to username / password authentication for both HTTP and SASL (i.e. Remoting) - both are being covered at the same time as predominantly they require the same core configuration, it is not until the definition of the authentication factories that the configuration becomes really specific.

The WildFly Elytron project already contains a dummy certificate authority set up that we use for testing, the KeyStores within this documentation are from the dummy certificate authority although for anything other than testing these should be replaced with your own.

As a first step we define some paths that point to the wildfly-elytron project so we can use these in the subsequent configuration.

```
./path=elytron.project:add(path=/home/darranl/src/wildfly10/wildfly-elytron)
./path=elytron.project.jks:add(path=src/test/resources/ca/jks, relative-to=elytron.project)
./path=elytron.project.properties:add(path=src/test/resources/org/wildfly/security/auth/realm,
relative-to=elytron.project)
```

This results in the following configuration.

```
<paths>
    <path name="elytron.project" path="/home/darranl/src/wildfly10/wildfly-elytron"/>
    <path name="elytron.project.jks" path="src/test/resources/ca/jks"
relative-to="elytron.project"/>
    <path name="elytron.project.properties"
path="src/test/resources/org/wildfly/security/auth/realm" relative-to="elytron.project"/>
  </paths>
```

KeyStores, KeyManagers, and TrustManagers.

The next step is to define the KeyStore resources, for this example three different keystores are used: -

1. **localhost.keystore** - Contains the servers key and certificate for 'localhost'.
2. **beetles.keystore** - Contains the individual client certificates.
3. **ca.keystore** - Contains the certificate of the certificate authority.

When we define the overall configuration we will use the localhost keystore along with the ca keystore for the incoming connections so initially all client certificates signed by the certificate authority will be accepted and subsequently a security realm will check it against the actual certificates within beetles keystore.

```
./subsystem=elytron/key-store=localhost:add(type=jks, relative-to=elytron.project.jks,
path=localhost.keystore, credential-reference={clear-text=Elytron})
./subsystem=elytron/key-store=beetles:add(type=jks, relative-to=elytron.project.jks,
path=beetles.keystore, credential-reference={clear-text=Elytron})
./subsystem=elytron/key-store=ca:add(type=jks, relative-to=elytron.project.jks,
path=ca.truststore, credential-reference={clear-text=Elytron})
```

This results in the following definitions: -

```
<subsystem xmlns="urn:wildfly:elytron:1.1" final-providers="combined-providers"
disallowed-providers="OracleUcrypto">
    ...
    <tls>
      <key-stores>
        <key-store name="localhost">
          <credential-reference clear-text="Elytron"/>
          <implementation type="jks"/>
          <file path="localhost.keystore" relative-to="elytron.project.jks"/>
        </key-store>
        <key-store name="beetles">
          <credential-reference clear-text="Elytron"/>
          <implementation type="jks"/>
          <file path="beetles.keystore" relative-to="elytron.project.jks"/>
        </key-store>
        <key-store name="ca">
          <credential-reference clear-text="Elytron"/>
          <implementation type="jks"/>
          <file path="ca.truststore" relative-to="elytron.project.jks"/>
        </key-store>
      </key-stores>
    </tls>
  </subsystem>
```

Next the key and trust manager resources will be defined using these keystores.

```
./subsystem=elytron/key-manager=localhost-manager:add(algorithm=SunX509, key-store=localhost,
credential-reference={clear-text=Elytron})
./subsystem=elytron/trust-manager=ca-manager:add(algorithm=SunX509, key-store=ca)
```

Resulting in: -

```
<subsystem xmlns="urn:wildfly:elytron:1.1" final-providers="combined-providers"
disallowed-providers="OracleUcrypto">
    ...
    <tls>
     ...
       <key-managers>
         <key-manager name="localhost-manager" algorithm="SunX509" key-store="localhost">
           <credential-reference clear-text="Elytron"/>
         </key-manager>
       </key-managers>
       <trust-managers>
         <trust-manager name="ca-manager" algorithm="SunX509" key-store="ca"/>
       </trust-managers>
    </tls>
  </subsystem>
```

Realms and Domains

Two security realms are now defined, one of these uses properties files from within the WildFly Elytron project to support username/password authentication and the other using the clients certificates for verification.

```
./subsystem=elytron/properties-realm=test-users:add(users-properties={relative-to=elytron.project.
path=clear.properties, plain-text=true, digest-realm-name=ManagementRealm},
groups-properties={relative-to=elytron.project.properties, path=groups.properties})
./subsystem=elytron/key-store-realm=key-store-realm:add(key-store=beetles)
```

Resulting in: -

```
<subsystem xmlns="urn:wildfly:elytron:1.1" final-providers="combined-providers"
disallowed-providers="OracleUcrypto">
    ...
    <security-realms>
      ...
      <key-store-realm name="key-store-realm" key-store="beetles"/>
      ...
      <properties-realm name="test-users">
        <users-properties path="clear.properties" relative-to="elytron.project.properties"
digest-realm-name="ManagementRealm" plain-text="true"/>
        <groups-properties path="groups.properties" relative-to="elytron.project.properties"/>
      </properties-realm>
    </security-realms>
    ...
  </subsystem>
```

These security realms can now be referenced from a security domain: -

```
./subsystem=elytron/security-domain=client-cert-domain:add(realms=[{realm=test-users},{realm=key-s
\
  default-realm=test-users, \
  permission-mapper=default-permission-mapper)
```

Resulting in: -

```
<subsystem xmlns="urn:wildfly:elytron:1.1" final-providers="combined-providers"
disallowed-providers="OracleUcrypto">
    ...
    <security-domains>
      ...
      <security-domain name="client-cert-domain" default-realm="test-users"
permission-mapper="default-permission-mapper">
        <realm name="test-users"/>
        <realm name="key-store-realm"/>
      </security-domain>
    </security-domains>
    ...
  </subsystem>
```

Before moving onto the individual authentication factories a couple of additional utility resources are also required: -

```
./subsystem=elytron/constant-realm-mapper=key-store-realm:add(realm-name=key-store-realm)
./subsystem=elytron/x500-attribute-principal-decoder=x500-decoder:add(attribute-name=CN,
maximum-segments=1)
```

Resulting in: -

```
<subsystem xmlns="urn:wildfly:elytron:1.1" final-providers="combined-providers"
disallowed-providers="OracleUcrypto">
    ...
    <mappers>
      ...
      <x500-attribute-principal-decoder name="x500-decoder" attribute-name="CN"
maximum-segments="1"/>
      ...
      <constant-realm-mapper name="key-store-realm" realm-name="key-store-realm"/>
      ...
    </mappers>
    ...
  </subsystem>
```

HTTP Authentication Factory

For the HTTP connections we now define a HTTP authentication factory using the previously defined resources and it is configured to support CLIENT_CERT and DIGEST authentication.

```
./subsystem=elytron/http-authentication-factory=client-cert-digest:add(http-server-mechanism-facto
\
  security-domain=client-cert-domain, \
 mechanism-configurations=[{ \
  mechanism-name=CLIENT_CERT, \
  realm-mapper=key-store-realm, \
  pre-realm-principal-transformer=x500-decoder}, \
 {mechanism-name=DIGEST, mechanism-realm-configurations=[{realm-name=ManagementRealm}]}])
```

Resulting in: -

```
<subsystem xmlns="urn:wildfly:elytron:1.1" final-providers="combined-providers"
disallowed-providers="OracleUcrypto">
    ...
    <http>
      ...
      <http-authentication-factory name="client-cert-digest"
http-server-mechanism-factory="global" security-domain="client-cert-domain">
        <mechanism-configuration>
          <mechanism mechanism-name="CLIENT_CERT" pre-realm-principal-transformer="x500-decoder"
realm-mapper="key-store-realm"/>
          <mechanism mechanism-name="DIGEST">
            <mechanism-realm realm-name="ManagementRealm"/>
          </mechanism>
        </mechanism-configuration>
      </http-authentication-factory>
      ...
    </http>
    ...
  </subsystem>
```

Where DIGEST authentication is used we rely on the default configuration within the security domain to select the 'test-users' realm, however where CLIENT_CERT authentication is in use an alternative realm-mapper is referenced to ensure the 'key-store-realm' is used.

Additionally for CLIENT_CERT authentication a principal-transformer is referenced to extract the CN attribute from the distinguished name of the client certificate and use this when accessing the identity from the security realm.

SASL Authentication Factory

The architecture of the two authentication factories if very similar so a SASL authentication factory can be defined in the same way as the HTTP equivalent.

```
./subsystem=elytron/sasl-authentication-factory=client-cert-digest:add(sasl-server-factory=elytron
\
  security-domain=client-cert-domain, \
  mechanism-configurations=[{mechanism-name=EXTERNAL, \
  realm-mapper=key-store-realm, \
  pre-realm-principal-transformer=x500-decoder}, \
  {mechanism-name=DIGEST-MD5, mechanism-realm-configurations=[{realm-name=ManagementRealm}]}])
```

This results in: -

```
<subsystem xmlns="urn:wildfly:elytron:1.1" final-providers="combined-providers"
disallowed-providers="OracleUcrypto">
    ...
    <sasl>
      ...
      <sasl-authentication-factory name="client-cert-digest" sasl-server-factory="elytron"
security-domain="client-cert-domain">
        <mechanism-configuration>
          <mechanism mechanism-name="EXTERNAL" pre-realm-principal-transformer="x500-decoder"
realm-mapper="key-store-realm"/>
          <mechanism mechanism-name="DIGEST-MD5">
            <mechanism-realm realm-name="ManagementRealm"/>
          </mechanism>
        </mechanism-configuration>
      </sasl-authentication-factory>
      ...
    </sasl>
    ...
  </subsystem>
```

Realm mappers and principal transformers are defined in the same way as were defined for HTTP.

SSL Context

An SSL context is also defined for use by the server.

```
./subsystem=elytron/server-ssl-context=localhost:add(key-manager=localhost-manager,
trust-manager=ca-manager, \
  security-domain=client-cert-domain, \
  authentication-optional=true, \
  want-client-auth=true, \
  need-client-auth=false)
```

Resulting in: -

```
<subsystem xmlns="urn:wildfly:elytron:1.1" final-providers="combined-providers"
disallowed-providers="OracleUcrypto">
    ...
    <tls>
      ...
      <server-ssl-contexts>
        <server-ssl-context name="localhost" security-domain="client-cert-domain"
want-client-auth="true" need-client-auth="false" authentication-optional="true"
key-manager="localhost-manager" trust-manager="ca-manager"/>
      </server-ssl-contexts>
    </tls>
  </subsystem>
```

As we will be supporting fallback to username/password authentication *need-client-auth* is set to *false* as well as *authentication-optional* being set to *false*, this allows connections to be established but an alternative form of authentication will be required.

Using for Management

At this point the management interfaces can be updated to use the newly defined resources, we need to add references to the two new authentication factories and the SSL context, we can also remove the existing reference to the legacy security realm. As this is modifying existing interfaces a server reload will also be required.

```
./core-service=management/management-interface=http-interface:write-attribute(name=ssl-context,
value=localhost)
./core-service=management/management-interface=http-interface:write-attribute(name=secure-socket-b
value=management-https)
./core-service=management/management-interface=http-interface:write-attribute(name=http-authentica
value=client-cert-digest)
./core-service=management/management-interface=http-interface:write-attribute(name=http-upgrade.sa
value=client-cert-digest)
./core-service=management/management-interface=http-interface:undefine-attribute(name=security-rea
```

The management interface configuration then becomes: -

```
<management>
    ...
    <management-interfaces>
      <http-interface http-authentication-factory="client-cert-digest" ssl-context="localhost">
        <http-upgrade enabled="true" sasl-authentication-factory="client-cert-digest"/>
        <socket-binding http="management-http" https="management-https"/>
      </http-interface>
    </management-interfaces>
    ...
  </management>
```

Admin Clients

At this stage assuming the same files have been used as in this example it should be possible to connect to the management interface of the server either using a web browser or the JBoss CLI with the username *elytron* and password *passwd12#$*

For certificate based authentication the keys and certificates from the WildFly Elytron tests can be used, these are found in JKS keystores under 'src/test/resources/ca/jks', these keystores have a password of *Elytron*.

Web Browser Configuration
A PKCS#12 file can be created from the test keystores,this can then be imported into the web browser to use when connecting to the server.

```
keytool -importkeystore -srckeystore ladybird.keystore \
  -destkeystore ladybird.pkcs12 \
  -srcstoretype jks \
  -deststoretype pkcs12 \
  -deststorepass Elytron \
  -srcalias ladybird \
  -destalias ladybird
```

CLI Configuration

Since the integration of WildFly Elytron it is possible with the CLI to use a configuration file wildfly-config.xml to define the security settings including the settings for the client side SSL context.

For the purpose of this example copy the ladybird,keystore and ca.truststore from the Wildfly Elytron testsuite to the location the JBoss CLI is being started from, the following wildfly-config.xml can be created in this location as well: -

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration>
    <authentication-client xmlns="urn:elytron:1.0">
        <key-stores>
            <key-store name="ladybird" type="jks" >
                <file name="ladybird.keystore"/>
                <key-store-clear-password password="Elytron" />
            </key-store>
            <key-store name="ca" type="jks">
                <file name="ca.truststore"/>
                <key-store-clear-password password="Elytron" />
            </key-store>
        </key-stores>
        <ssl-context-rules>
            <rule use-ssl-context="default" />
        </ssl-context-rules>
        <ssl-contexts>
            <ssl-context name="default">
                <key-store-ssl-certificate key-store-name="ladybird" alias="ladybird">
                    <key-store-clear-password password="Elytron" />
                </key-store-ssl-certificate>
                <trust-store key-store-name="ca" />
            </ssl-context>
        </ssl-contexts>
    </authentication-client>
</configuration>
```

The CLI can now be started using the following command: -

```
./jboss-cli.sh -c -Dwildfly.config.url=wildfly-config.xml
```

The *:whoami* command can be used within the CLI to double check the current identity.

```
[standalone@localhost:9993 /] :whoami(verbose=true)
{
    "outcome" => "success",
    "result" => {
        "identity" => {"username" => "Ladybird"},
        "mapped-roles" => ["SuperUser"]
    }
}
```

15.11.5 Documentation Still Needed

- How to migrate application which uses different identity store for authentication and authorization (migration to Elytron aggregate-realm).
- How migrate to using cache (migration to caching-realm)
- Limitations for migration from PicketBox/legacy security to Elytron, for example, Infinispan cache cannot be used, any others?

15.11.6 Application Client Migration

Naming Client

This migration example assumes a client application performs a remote JNDI lookup using an `InitialContext` backed by the `org.jboss.naming.remote.client.InitialContextFactory` class.

Original Configuration

An `InitialContext` backed by the `org.jboss.naming.remote.client.InitialContextFactory` class can be created by specifying properties that contain the URL of the naming provider to connect to along with appropriate user credentials:

```
Properties properties = new Properties();
properties.put(Context.INITIAL_CONTEXT_FACTORY,
"org.jboss.naming.remote.client.InitialContextFactory");
properties.put(Context.PROVIDER_URL, "http-remoting://127.0.0.1:8080");
properties.put(Context.SECURITY_PRINCIPAL, "bob");
properties.put(Context.SECURITY_CREDENTIALS, "secret");
InitialContext context = new InitialContext(properties);
Bar bar = (Bar) context.lookup("foo/bar");
...
```

Migrated Configuration

An `InitialContext` backed by the `org.wildfly.naming.client.WildFlyInitialContextFactory` class can be created by specifying a property that contains the URL of the naming provider to connect to. The user credentials can be specified using a WildFly client configuration file or programmatically.

Configuration File Approach

A `wildfly-config.xml` file that contains the user credentials to use when establishing a connection to the naming provider can be added to the client application's `META-INF` directory:

wildfly-config.xml

```
<configuration>
    <authentication-client xmlns="urn:elytron:1.0">
        <authentication-rules>
            <rule use-configuration="namingConfig">
                <match-host name="127.0.0.1"/>
            </rule>
        </authentication-rules>
        <authentication-configurations>
            <configuration name="namingConfig">
                <set-user-name name="bob"/>
                <credentials>
                    <clear-password password="secret"/>
                </credentials>
            </configuration>
        </authentication-configurations>
    </authentication-client>
</configuration>
```

An `InitialContext` can then be created as follows:

```
Properties properties = new Properties();
properties.put(Context.INITIAL_CONTEXT_FACTORY,
"org.wildfly.naming.client.WildFlyInitialContextFactory");
properties.put(Context.PROVIDER_URL, "remote+http://127.0.0.1:8080");
InitialContext context = new InitialContext(properties);
Bar bar = (Bar) context.lookup("foo/bar");
...
```

Programmatic Approach

The user credentials to use when establishing a connection to the naming provider can be specified directly in the client application's code:

```
// create your authentication configuration
AuthenticationConfiguration namingConfig =
AuthenticationConfiguration.empty().useName("bob").usePassword("secret");

// create your authentication context
AuthenticationContext context =
AuthenticationContext.empty().with(MatchRule.ALL.matchHost("127.0.0.1"), namingConfig);

// create a callable that creates and uses an InitialContext
Callable<Void> callable = () -> {
    Properties properties = new Properties();
    properties.put(Context.INITIAL_CONTEXT_FACTORY,
"org.wildfly.naming.client.WildFlyInitialContextFactory");
    properties.put(Context.PROVIDER_URL, "remote+http://127.0.0.1:8080");
    InitialContext context = new InitialContext(properties);
    Bar bar = (Bar) context.lookup("foo/bar");
    ...
    return null;
};

// use your authentication context to run your callable
context.runCallable(callable);
```

EJB Client

This migration example assumes a client application is configured to invoke an EJB deployed on a remote server using a `jboss-ejb-client.properties` file.

Original Configuration

A `jboss-ejb-client.properties` file that contains the information needed to connect to the remote server can be specified in a client application's `META-INF` directory:

jboss-ejb-client.properties

```
remote.connectionprovider.create.options.org.xnio.Options.SSL_ENABLED=false
remote.connections=default
remote.connection.default.host=127.0.0.1
remote.connection.default.port = 8080
remote.connection.default.username=bob
remote.connection.default.password=secret
```

An EJB can then be looked up and a method can be invoked on it as follows:

```
// create an InitialContext
Properties properties = new Properties();
properties.put(Context.INITIAL_CONTEXT_FACTORY,
"org.jboss.naming.remote.client.InitialContextFactory");
properties.put(Context.URL_PKG_PREFIXES, "org.jboss.ejb.client.naming");
InitialContext context = new InitialContext(properties);

// look up an EJB and invoke one of its methods
RemoteCalculator statelessRemoteCalculator = (RemoteCalculator) context.lookup(
    "ejb:/ejb-remote-server-side//CalculatorBean!" + RemoteCalculator.class.getName());
int sum = statelessRemoteCalculator.add(101, 202);
```

Migrated Configuration

The information needed to connect to the remote server can be specified using a WildFly client configuration file or programmatically.

Configuration File Approach

A `wildfly-config.xml` file that contains the information needed to connect to the remote server can be added to the client application's `META-INF` directory:

wildfly-config.xml

```
<configuration>
    <authentication-client xmlns="urn:elytron:1.0">
        <authentication-rules>
            <rule use-configuration="ejbConfig">
                <match-host name="127.0.0.1"/>
            </rule>
        </authentication-rules>
        <authentication-configurations>
            <configuration name="ejbConfig">
                <set-user-name name="bob"/>
                <credentials>
                    <clear-password password="secret"/>
                </credentials>
            </configuration>
        </authentication-configurations>
    </authentication-client>
    <jboss-ejb-client xmlns="urn:jboss:wildfly-client-ejb:3.0">
        <connections>
            <connection uri="remote+http://127.0.0.1:8080" />
        </connections>
    </jboss-ejb-client>
</configuration>
```

An EJB can then be looked up and a method can be invoked on it as follows:

```
// create an InitialContext
Properties properties = new Properties();
properties.put(Context.INITIAL_CONTEXT_FACTORY,
"org.wildfly.naming.client.WildFlyInitialContextFactory");
InitialContext context = new InitialContext(properties);

// look up an EJB and invoke one of its methods (same as before)
RemoteCalculator statelessRemoteCalculator = (RemoteCalculator) context.lookup(
    "ejb:/ejb-remote-server-side//CalculatorBean!" + RemoteCalculator.class.getName());
int sum = statelessRemoteCalculator.add(101, 202);
```

Programmatic Approach

The information needed to connect to the remote server can be specified directly in the client application's code:

```
// create your authentication configuration
AuthenticationConfiguration ejbConfig =
AuthenticationConfiguration.empty().useName("bob").usePassword("secret");

// create your authentication context
AuthenticationContext context =
AuthenticationContext.empty().with(MatchRule.ALL.matchHost("127.0.0.1"), ejbConfig);

// create a callable that invokes an EJB
Callable<Void> callable = () -> {

    // create an InitialContext
    Properties properties = new Properties();
    properties.put(Context.INITIAL_CONTEXT_FACTORY,
"org.wildfly.naming.client.WildFlyInitialContextFactory");
    properties.put(Context.PROVIDER_URL, "remote+http://127.0.0.1:8080");
    InitialContext context = new InitialContext(properties);

    // look up an EJB and invoke one of its methods (same as before)
    RemoteCalculator statelessRemoteCalculator = (RemoteCalculator) context.lookup(
        "ejb:/ejb-remote-server-side//CalculatorBean!" + RemoteCalculator.class.getName());
    int sum = statelessRemoteCalculator.add(101, 202);
    ...
    return null;
};

// use your authentication context to run your callable
context.runCallable(callable);
```

15.11.7 Caching Migration

Where a PicketBox based security domain is defined it is possible to enable caching for that security domain, this enables subsequent hits to the identity store to be avoided as an in memory cache can be used instead, this example demonstrates how caching can be used with a WildFly Elytron based configuration.

The purpose of this chapter is to highlight the migration of a configuration with caching enabled, this example is based in the previous LDAP example but with caching enabled.

PicketBox Example

A PicketBox based security domain can be defined with the following commands.

```
./subsystem=security/security-domain=application-security:add(cache-type=default)
./subsystem=security/security-domain=application-security/authentication=classic:add(login-modules=
flag=Required, module-options={ \
java.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory, \
java.naming.provider.url=ldap://localhost:10389, \
java.naming.security.authentication=simple, \
bindDN="uid=admin,ou=system", \
bindCredential=secret, \
baseCtxDN="ou=users,dc=group-to-principal,dc=wildfly,dc=org", \
baseFilter="(uid={0})", \
rolesCtxDN="ou=groups,dc=group-to-principal,dc=wildfly,dc=org",\
roleFilter="(uniqueMember={1})", \
roleAttributeID="uid" \
}])
```

Resulting in the following security domain definition: -

```
<subsystem xmlns="urn:jboss:domain:security:2.0">
    <security-domains>
      ...
      <security-domain name="application-security" cache-type="default">
        <authentication>
          <login-module code="LdapExtended" flag="required">
            <module-option name="java.naming.factory.initial"
value="com.sun.jndi.ldap.LdapCtxFactory"/>
            <module-option name="java.naming.provider.url" value="ldap://localhost:10389"/>
            <module-option name="java.naming.security.authentication" value="simple"/>
            <module-option name="bindDN" value="uid=admin,ou=system"/>
            <module-option name="bindCredential" value="secret"/>
            <module-option name="baseCtxDN"
value="ou=users,dc=group-to-principal,dc=wildfly,dc=org"/>
            <module-option name="baseFilter" value="(uid={0})"/>
            <module-option name="rolesCtxDN"
value="ou=groups,dc=group-to-principal,dc=wildfly,dc=org"/>
            <module-option name="roleFilter" value="(uniqueMember={1})"/>
            <module-option name="roleAttributeID" value="uid"/>
          </login-module>
        </authentication>
      </security-domain>
    </security-domains>
  </subsystem>
```

The key difference to the raw LDAP example is that a cache-type of 'default' has been specified on the security domain. The default cache-type is an in memory cache, when using PicketBox it is also possible to specify a cache-type of 'infinispan' although this is not supported with WildFly Elytron as various aspects of a SecurityIdentity are not suitable for replication.

Migrated Example

When using WildFly Elytron where caching is required the individual security realm is wrapped using a cache, a migrated configuration can be defined with the following commands: -

```
./subsystem=elytron/dir-context=ldap-connection:add(url=ldap://localhost:10389,
principal="uid=admin,ou=system", credential-reference={clear-text=secret})
./subsystem=elytron/ldap-realm=ldap-realm:add(dir-context=ldap-connection, \
direct-verification=true, \
identity-mapping={search-base-dn="ou=users,dc=group-to-principal,dc=wildfly,dc=org", \
rdn-identifier="uid", \
attribute-mapping=[{filter-base-dn="ou=groups,dc=group-to-principal,dc=wildfly,dc=org",filter="(un
```

These can then be used in a security domain and subsequently an authentication factory.

```
./subsystem=elytron/security-domain=application-security:add(realms=[{realm=cached-ldap}],
default-realm=cached-ldap, permission-mapper=default-permission-mapper)
./subsystem=elytron/http-authentication-factory=application-security-http:add(http-server-mechanis
security-domain=application-security, mechanism-configurations=[{mechanism-name=BASIC}])
```

In this final step it is very important that the caching-realm is referenced rather than the original realm otherwise caching will be bypassed.

This results in the following definitions: -

```
<subsystem xmlns="urn:wildfly:elytron:1.1" final-providers="combined-providers"
disallowed-providers="OracleUcrypto">
    ...
    <security-domains>
      ...
      <security-domain name="application-security" default-realm="cached-ldap"
permission-mapper="default-permission-mapper">
        <realm name="cached-ldap"/>
      </security-domain>
    </security-domains>
    <security-realms>
      ...
      <ldap-realm name="ldap-realm" dir-context="ldap-connection" direct-verification="true">
        <identity-mapping rdn-identifier="uid"
search-base-dn="ou=users,dc=group-to-principal,dc=wildfly,dc=org">
          <attribute-mapping>
            <attribute from="uid" to="Roles" filter="(uniqueMember={1})"
filter-base-dn="ou=groups,dc=group-to-principal,dc=wildfly,dc=org"/>
          </attribute-mapping>
        </identity-mapping>
      </ldap-realm>
      <caching-realm name="cached-ldap" realm="ldap-realm"/>
    </security-realms>
    ...
    <http>
      ...
      <http-authentication-factory name="application-security-http"
http-server-mechanism-factory="global" security-domain="application-security">
        <mechanism-configuration>
          <mechanism mechanism-name="BASIC"/>
        </mechanism-configuration>
      </http-authentication-factory>
      ...
    </http>
    ...
    <dir-contexts>
      <dir-context name="ldap-connection" url="ldap://localhost:10389"
principal="uid=admin,ou=system">
        <credential-reference clear-text="secret"/>
      </dir-context>
    </dir-contexts>
  </subsystem>
```

15.11.8 Composite Stores Migration

When using either PicketBox or the legacy security realms it is possible to define a configuration where authentication is performed against one identity store whilst the information used for authorization is loaded from a different store, when using WildFly Elytron this can be achieved by using an aggregate security realm.

The example here makes use of a properties file for authentication and then searches LDAP to load group / role information. Both of these are based on the previous examples within this document so the environmental information is not repeated here.

PicketBox Based Configuration

A PicketBox based security domain can be created by using the following CLI commands: -

```
./subsystem=security/security-domain=application-security:add
./subsystem=security/security-domain=application-security/authentication=classic:add(login-modules=
\
{code=UsersRoles, flag=Required, module-options={ \
password-stacking=useFirstPass, \
usersProperties=file://${jboss.server.config.dir}/example-users.properties, \
rolesProperties=file://${jboss.server.config.dir}/example-roles.properties}} \
{code=LdapExtended, flag=Required, module-options={ \
password-stacking=useFirstPass, \
java.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory, \
java.naming.provider.url=ldap://localhost:10389, \
java.naming.security.authentication=simple, \
bindDN="uid=admin,ou=system", \
bindCredential=secret, \
baseCtxDN="ou=users,dc=group-to-principal,dc=wildfly,dc=org", \
baseFilter="(uid={0})", \
rolesCtxDN="ou=groups,dc=group-to-principal,dc=wildfly,dc=org",\
roleFilter="(uniqueMember={1})", \
roleAttributeID="uid" \
}}])
```

This results in the following domain definition: -

```
<security-domain name="application-security">
    <authentication>
      <login-module code="UsersRoles" flag="required">
        <module-option name="password-stacking" value="useFirstPass"/>
        <module-option name="usersProperties"
value="file://${jboss.server.config.dir}/example-users.properties"/>
        <module-option name="rolesProperties"
value="file://${jboss.server.config.dir}/example-roles.properties"/>
      </login-module>
      <login-module code="LdapExtended" flag="required">
        <module-option name="password-stacking" value="useFirstPass"/>
        <module-option name="java.naming.factory.initial"
value="com.sun.jndi.ldap.LdapCtxFactory"/>
        <module-option name="java.naming.provider.url" value="ldap://localhost:10389"/>
        <module-option name="java.naming.security.authentication" value="simple"/>
        <module-option name="bindDN" value="uid=admin,ou=system"/>
        <module-option name="bindCredential" value="secret"/>
        <module-option name="baseCtxDN"
value="ou=users,dc=group-to-principal,dc=wildfly,dc=org"/>
        <module-option name="baseFilter" value="(uid={0})"/>
        <module-option name="rolesCtxDN"
value="ou=groups,dc=group-to-principal,dc=wildfly,dc=org"/>
        <module-option name="roleFilter" value="(uniqueMember={1})"/>
        <module-option name="roleAttributeID" value="uid"/>
      </login-module>
    </authentication>
  </security-domain>
```

During an authentication attempt the 'UsersRoles' login module will first be called to perform authentication based on the supplied credential, then the 'LdapExtLoginModule' will be called which will proceed to query LDAP to load the roles for the identity.

Legacy Security Realm Configuration

An equivalent configuration can also be created using the legacy security realms with the following commands: -

```
./core-service=management/ldap-connection=MyLdapConnection:add(url="ldap://localhost:10389",
search-dn="uid=admin,ou=system", search-credential="secret")

./core-service=management/security-realm=ApplicationSecurity:add
./core-service=management/security-realm=ApplicationSecurity/authentication=properties:add(path=ex
relative-to=jboss.server.config.dir, plain-text=true)

batch
./core-service=management/security-realm=ApplicationSecurity/authorization=ldap:add(connection=MyL
base-dn="ou=users,dc=group-to-principal,dc=wildfly,dc=org")
./core-service=management/security-realm=ApplicationSecurity/authorization=ldap/group-search=group
iterative=true, prefer-original-connection=true, principal-attribute=uniqueMember,
search-by=DISTINGUISHED_NAME, group-name=SIMPLE, group-name-attribute=uid)
run-batch
```

This results in the following realm definition: -

```
<security-realm name="ApplicationSecurity">
    <authentication>
      <properties path="example-users.properties" relative-to="jboss.server.config.dir"
plain-text="true"/>
    </authentication>
    <authorization>
      <ldap connection="MyLdapConnection">
        <username-to-dn>
          <username-filter base-dn="ou=users,dc=group-to-principal,dc=wildfly,dc=org"
attribute="uid"/>
        </username-to-dn>
        <group-search group-name="SIMPLE" iterative="true" group-name-attribute="uid">
          <group-to-principal search-by="DISTINGUISHED_NAME"
base-dn="ou=groups,dc=group-to-principal,dc=wildfly,dc=org" prefer-original-connection="true">
            <membership-filter principal-attribute="uniqueMember"/>
          </group-to-principal>
        </group-search>
      </ldap>
    </authorization>
  </security-realm>

  <outbound-connections>
    <ldap name="MyLdapConnection" url="ldap://localhost:10389" search-dn="uid=admin,ou=system"
search-credential="secret"/>
  </outbound-connections>
```

As with the PicketBox example, authentication is first performed using the properties file - then group searching is performed against LDAP.

Migrated WildFly Elytron Configuration

The equivalent WildFly Elytron configuration can be defined with the following commands: -

```
./subsystem=elytron/dir-context=ldap-connection:add(url=ldap://localhost:10389,
principal="uid=admin,ou=system", credential-reference={clear-text=secret})

./subsystem=elytron/ldap-realm=ldap-realm:add(dir-context=ldap-connection, \
direct-verification=true, \
identity-mapping={search-base-dn="ou=users,dc=group-to-principal,dc=wildfly,dc=org", \
rdn-identifier="uid", \
attribute-mapping=[{filter-base-dn="ou=groups,dc=group-to-principal,dc=wildfly,dc=org",filter="(un
relative-to=jboss.server.config.dir, plain-text=true, digest-realm-name="Application Security"},
groups-properties={path=example-roles.properties, relative-to=jboss.server.config.dir},
groups-attribute=Roles)

./subsystem=elytron/aggregate-realm=combined-realm:add(authentication-realm=application-properties
authorization-realm=ldap-realm)

./subsystem=elytron/security-domain=application-security:add(realms=[{realm=combined-realm}],
default-realm=combined-realm, permission-mapper=default-permission-mapper)
./subsystem=elytron/http-authentication-factory=application-security-http:add(http-server-mechanis
security-domain=application-security, mechanism-configurations=[{mechanism-name=BASIC}])
```

This results in the following definitions: -

```
<subsystem xmlns="urn:wildfly:elytron:1.1" final-providers="combined-providers"
disallowed-providers="OracleUcrypto">
    ...
    <security-domains>
      ...
      <security-domain name="application-security" default-realm="combined-realm"
permission-mapper="default-permission-mapper">
        <realm name="combined-realm"/>
      </security-domain>
    </security-domains>
    <security-realms>
      <aggregate-realm name="combined-realm" authentication-realm="application-properties"
authorization-realm="ldap-realm"/>
        ...
        <properties-realm name="application-properties" groups-attribute="Roles">
          <users-properties path="example-users.properties"
relative-to="jboss.server.config.dir" digest-realm-name="Application Security"
plain-text="true"/>
          <groups-properties path="example-roles.properties"
relative-to="jboss.server.config.dir"/>
        </properties-realm>
        <ldap-realm name="ldap-realm" dir-context="ldap-connection" direct-verification="true">
          <identity-mapping rdn-identifier="uid"
search-base-dn="ou=users,dc=group-to-principal,dc=wildfly,dc=org">
            <attribute-mapping>
              <attribute from="uid" to="Roles" filter="(uniqueMember={1})"
filter-base-dn="ou=groups,dc=group-to-principal,dc=wildfly,dc=org"/>
            </attribute-mapping>
          </identity-mapping>
        </ldap-realm>
    </security-realms>
    ...
    <http>
      ...
      <http-authentication-factory name="application-security-http"
http-server-mechanism-factory="global" security-domain="application-security">
        <mechanism-configuration>
          <mechanism mechanism-name="BASIC"/>
        </mechanism-configuration>
      </http-authentication-factory>
      ...
    </http>
    ...
    <dir-contexts>
      <dir-context name="ldap-connection" url="ldap://localhost:10389"
principal="uid=admin,ou=system">
        <credential-reference clear-text="secret"/>
      </dir-context>
    </dir-contexts>
  </subsystem>
```

Within the WildFly Elytron example a new security realm 'aggregate-realm' has been defined, this definition specifies which of the defined security realms should be used for the authentication step and which of the security realms should be used for the loading of the identity used for subsequent authorization decisions.

15.11.9 Database Authentication

The section describing how to migrate from database accessible via JDBC datasource based authentication using PicketBox to Elytron. This section will illustrate some equivalent configuration using PicketBox security domains and show the equivalent configuration using Elytron but will not repeat the steps to wire it all together covered in the previous sections.

These configuration examples are developed against a test database with users table like:

```
CREATE TABLE User (
    id BIGINT NOT NULL,
    username VARCHAR(255),
    password VARCHAR(255),
    role ENUM('admin', 'manager', 'user'),
    PRIMARY KEY (id),
    UNIQUE (username)
)
```

For authentication purposes the username will be matched against the '`username`' column, password will be expected in hex-encoded MD5 hash in '`password`' column. User role for authorization purposes will be taken from '`role`' column.

PicketBox Database LoginModule

The following commands can create a PicketBox security domain configured to use database accessible via JDBC datasource to verify a username and password and to assign roles.

```
./subsystem=security/security-domain=application-security/:add
./subsystem=security/security-domain=application-security/authentication=classic:add(login-modules=
flag=Required, module-options={ \
    dsJndiName="java:jboss/datasources/ExampleDS", \
    principalsQuery="SELECT password FROM User WHERE username = ?", \
    rolesQuery="SELECT role, 'Roles' FROM User WHERE username = ?", \
    hashAlgorithm=MD5, \
    hashEncoding=base64 \
}}])
```

This results in the following configuration.

```
<subsystem xmlns="urn:jboss:domain:security:2.0">
            <security-domains>
                ...
                <security-domain name="application-security">
                    <authentication>
                        <login-module code="Database" flag="required">
                            <module-option name="dsJndiName"
value="java:jboss/datasources/ExampleDS"/>
                            <module-option name="principalsQuery" value="SELECT password FROM
User WHERE username = ?"/>
                            <module-option name="rolesQuery" value="SELECT role, 'Roles' FROM
User WHERE username = ?"/>
                            <module-option name="hashAlgorithm" value="MD5"/>
                            <module-option name="hashEncoding" value="base64"/>
                        </login-module>
                    </authentication>
                </security-domain>
            </security-domains>
        </subsystem>
```

Migrated

Within the Elytron subsystem to use database accesible via JDBC you need to define `jdbc-realm`:

```
./subsystem=elytron/jdbc-realm=jdbc-realm:add(principal-query=[{ \
    data-source=ExampleDS, \
    sql="SELECT role, password FROM User WHERE username = ?", \
    attribute-mapping=[{index=1, to=Roles}] \
    simple-digest-mapper={algorithm=simple-digest-md5, password-index=2}, \
}])
```

This results in the following overall configuration:

```
<subsystem xmlns="urn:wildfly:elytron:1.0" final-providers="combined-providers"
disallowed-providers="OracleUcrypto">
            ...
            <security-realms>
                ...
                <jdbc-realm name="jdbc-realm">
                    <principal-query sql="SELECT role, password FROM User WHERE username = ?"
data-source="ExampleDS">
                        <attribute-mapping>
                            <attribute to="Roles" index="1"/>
                        </attribute-mapping>
                        <simple-digest-mapper password-index="2"/>
                    </principal-query>
                </jdbc-realm>
                ...
            </security-realms>
            ...
        </subsystem>
```

In comparison with PicketBox solution, Elytron `jdbc-realm` use one SQL query to obtain all user attributes and credentials. Their extraction from SQL result specifies mappers.

N-M relation beetween user and roles

When using a n:m-relation beetween user and roles (which means: the user has multiple roles), the previous configuration does not work.

The database:

```
CREATE TABLE User (
    id BIGINT NOT NULL,
    username VARCHAR(255),
    password VARCHAR(255),
    PRIMARY KEY (id),
    UNIQUE (username)
)

CREATE TABLE Role(
    id BIGINT NOT NULL,
    rolename VARCHAR(255),
    PRIMARY KEY (id),
    UNIQUE (rolename)
)

CREATE TABLE Userrole(
    userid BIGINT not null,
    roleid BIGINT not null,
    PRIMARY KEY (userid, roleid),
    FOREIGN KEY (userid) references User(id,
    FOREIGN KEY (roleid) references Role(id)
)
```

Here you need two configure two principal queries:

```
<jdbc-realm name="jdbc-realm">
  <principal-query sql="SELECT PASSWORD FROM USER WHERE USERNAME = ?" data-source="ExampleDS">
   <clear-password-mapper password-index="1"/>
  </principal-query>
  <principal-query sql="SELECT R.ROLENAME from ROLE AS R, USERROLE AS UR, USER AS U WHERE
U.USERNAME=? AND UR.ROLEID = R.ID AND UR.USERID = U.ID" data-source="ExampleDS">
   <attribute-mapping>
    <attribute to="roles" index="1"/>
   </attribute-mapping>
  </principal-query>
 </jdbc-realm>
```

The second query needs an attribute mapping to decode the selected rolename column (index 1):

```
<mappers>
  ...
  <simple-role-decoder name="from-roles-attribute" attribute="roles"/>
  ...
 </mappers>
```

The role decoder is referenced by the security domain:

```
<security-domain name="MyDomain" default-realm="jdbc-realm"
permission-mapper="default-permission-mapper">
  <realm name="MyDbRealm" role-decoder="from-roles-attribute"/>
 </security-domain>
```

15.11.10 Kerberos Authentication Migration

When working with Kerberos configuration it is possible for the application server to rely on configuration from the environment or the key configuration can be specified using system properties, for the purpose of these examples I define system properties - these properties are applicable to both the legacy configuration and the migrated Elytron configuration.

```
./system-property=sun.security.krb5.debug:add(value=true)
./system-property=java.security.krb5.realm:add(value=ELYTRON.ORG)
./system-property=java.security.krb5.kdc:add(value=kdc.elytron.org)
```

The first line makes debugging easier but the last two lines specify the Kerberos realm in use and the address of the KDC.

```
<system-properties>
    <property name="sun.security.krb5.debug" value="true"/>
    <property name="java.security.krb5.realm" value="ELYTRON.ORG"/>
    <property name="java.security.krb5.kdc" value="kdc.elytron.org"/>
  </system-properties>
```

HTTP Authentication

Legacy Security Realm

A legacy security realm can be define so that SPNEGO authentication can be enabled for the HTTP management interface.

```
./core-service=management/security-realm=Kerberos:add
./core-service=management/security-realm=Kerberos/server-identity=kerberos:add
./core-service=management/security-realm=Kerberos/server-identity=kerberos/keytab=HTTP\/test-serve
debug=true)
./core-service=management/security-realm=Kerberos/authentication=kerberos:add(remove-realm=true)
```

This results in the following configuration: -

```
<security-realms>
    ...
    <security-realm name="Kerberos">
      <server-identities>
        <kerberos>
          <keytab principal="HTTP/test-server.elytron.org@ELYTRON.ORG"
path="/home/darranl/src/kerberos/test-server.keytab" debug="true"/>
        </kerberos>
      </server-identities>
      <authentication>
        <kerberos remove-realm="true"/>
      </authentication>
    </security-realm>
  </security-realms>
```

Application SPNEGO

Alternatively deployed applications would make use of a pair of security domains.

```
./subsystem=security/security-domain=host:add
./subsystem=security/security-domain=host/authentication=classic:add
./subsystem=security/security-domain=host/authentication=classic/login-module=1:add(code=Kerberos,
flag=Required, module-options={storeKey=true, useKeyTab=true,
principal=HTTP/test-server.elytron.org@ELYTRON.ORG,
keyTab=/home/darranl/src/kerberos/test-server.keytab, debug=true}
```

```
./subsystem=security/security-domain=SPNEGO:add
./subsystem=security/security-domain=SPNEGO/authentication=classic:add
./subsystem=security/security-domain=SPNEGO/authentication=classic/login-module=1:add(code=SPNEGO,
flag=requisite,  module-options={password-stacking=useFirstPass, serverSecurityDomain=host})
./subsystem=security/security-domain=SPNEGO/authentication=classic/login-module=1:write-attribute(
value=org.jboss.security.negotiation)
./subsystem=security/security-domain=SPNEGO/authentication=classic/login-module=2:add(code=UsersRo
flag=required, module-options={password-stacking=useFirstPass,
usersProperties=file:///home/darranl/src/kerberos/spnego-users.properties,
rolesProperties=file:///home/darranl/src/kerberos/spnego-roles.properties,
defaultUsersProperties=file:///home/darranl/src/kerberos/spnego-users.properties,
defaultRolesProperties=file:///home/darranl/src/kerberos/spnego-roles.properties})
```

This results in: -

```
<subsystem xmlns="urn:jboss:domain:security:2.0">
    <security-domains>
      ...
      <security-domain name="host">
        <authentication>
          <login-module name="1" code="Kerberos" flag="required">
            <module-option name="storeKey" value="true"/>
            <module-option name="useKeyTab" value="true"/>
            <module-option name="principal" value="HTTP/test-server.elytron.org@ELYTRON.ORG"/>
            <module-option name="keyTab" value="/home/darranl/src/kerberos/test-server.keytab"/>
            <module-option name="debug" value="true"/>
          </login-module>
        </authentication>
      </security-domain>
      <security-domain name="SPNEGO">
        <authentication>
          <login-module name="1" code="SPNEGO" flag="requisite"
module="org.jboss.security.negotiation">
            <module-option name="password-stacking" value="useFirstPass"/>
            <module-option name="serverSecurityDomain" value="host"/>
          </login-module>
          <login-module name="2" code="UsersRoles" flag="required">
            <module-option name="password-stacking" value="useFirstPass"/>
            <module-option name="usersProperties"
value="file:///home/darranl/src/kerberos/spnego-users.properties"/>
            <module-option name="rolesProperties"
value="file:///home/darranl/src/kerberos/spnego-roles.properties"/>
            <module-option name="defaultUsersProperties"
value="file:///home/darranl/src/kerberos/spnego-users.properties"/>
            <module-option name="defaultRolesProperties"
value="file:///home/darranl/src/kerberos/spnego-roles.properties"/>
          </login-module>
        </authentication>
      </security-domain>
    </security-domains>
  </subsystem>
```

An application can now be deployed referencing the SPNEGO security domain and secured with SPNEGO mechanism.

Migrated SPNEGO

The equivalent configuration can be achieved with WildFly Elytron by first defining a security realm which will be used to load identity information.

```
./subsystem=elytron/properties-realm=spnego-properties:add(users-properties={path=/home/darranl/sr
plain-text=true, digest-realm-name=ELYTRON.ORG},
groups-properties={path=/home/darranl/src/kerberos/spnego-roles.properties})
```

Next a Kerberos security factory is defined which allows the server to load it's own Kerberos identity.

```
./subsystem=elytron/kerberos-security-factory=test-server:add(path=/home/darranl/src/kerberos/test
principal=HTTP/test-server.elytron.org@ELYTRON.ORG, debug=true)
```

As with the previous examples we define a security realm to pull together the policy as well as a HTTP authentication factory for the authentication policy.

```
./subsystem=elytron/security-domain=SPNEGODomain:add(default-realm=spnego-properties,
realms=[{realm=spnego-properties, role-decoder=groups-to-roles}],
permission-mapper=default-permission-mapper)
./subsystem=elytron/http-authentication-factory=spnego-http-authentication:add(security-domain=SPN
http-server-mechanism-factory=global,mechanism-configurations=[{mechanism-name=SPNEGO,
credential-security-factory=test-server}])
```

Overall this results in the following configuration: -

```
<subsystem xmlns="urn:wildfly:elytron:1.0" final-providers="combined-providers"
disallowed-providers="OracleUcrypto">
    ...
    <security-domains>
    ...
      <security-domain name="SPNEGODomain" default-realm="spnego-properties"
permission-mapper="default-permission-mapper">
        <realm name="spnego-properties" role-decoder="groups-to-roles"/>
      </security-domain>
    </security-domains>
    <security-realms>
      ...
      <properties-realm name="spnego-properties">
        <users-properties path="/home/darranl/src/kerberos/spnego-users.properties"
digest-realm-name="ELYTRON.ORG" plain-text="true"/>
        <groups-properties path="/home/darranl/src/kerberos/spnego-roles.properties"/>
      </properties-realm>
    </security-realms>
    <credential-security-factories>
      <kerberos-security-factory name="test-server"
principal="HTTP/test-server.elytron.org@ELYTRON.ORG"
path="/home/darranl/src/kerberos/test-server.keytab" debug="true"/>
    </credential-security-factories>
    ...
    <http>
      ...
      <http-authentication-factory name="spnego-http-authentication"
http-server-mechanism-factory="global" security-domain="SPNEGODomain">
        <mechanism-configuration>
          <mechanism mechanism-name="SPNEGO" credential-security-factory="test-server"/>
        </mechanism-configuration>
      </http-authentication-factory>
      ...
    </http>
    ...
  </subsystem>
```

Now, to enable SPNEGO authentication for the HTTP management interface, update this interface to reference the `http-authentication-factory` defined above, as described in the properties authentication section.

Alternatively, to secure an application using SPNEGO authentication, an application security domain can be defined in the Undertow subsystem to map security domains to the `http-authentication-factory` defined above, as described in the properties authentication section.

Remoting / SASL Authentication

Legacy Security Realm

It is also possible to define a legacy security realm for Kerberos / GSSAPI SASL authenticatio for Remoting authentication such as the native management interface.

```
./core-service=management/security-realm=Kerberos:add
./core-service=management/security-realm=Kerberos/server-identity=kerberos:add
./core-service=management/security-realm=Kerberos/server-identity=kerberos/keytab=remote\/test-ser
debug=true)
./core-service=management/security-realm=Kerberos/authentication=kerberos:add(remove-realm=true)
```

```
<management>
    <security-realms>
      ...
      <security-realm name="Kerberos">
        <server-identities>
          <kerberos>
            <keytab principal="remote/test-server.elytron.org@ELYTRON.ORG"
path="/home/darranl/src/kerberos/remote-test-server.keytab" debug="true"/>
          </kerberos>
        </server-identities>
        <authentication>
          <kerberos remove-realm="true"/>
        </authentication>
      </security-realm>
    </security-realms>
    ...
  </management>
```

Migrated GSSAPI

The steps to define the equivalent Elytron configuration are very similar to the HTTP example.

First define the security realm to load the identity from: -

```
./path=kerberos:add(relative-to=user.home, path=src/kerberos)
./subsystem=elytron/properties-realm=kerberos-properties:add(users-properties={path=kerberos-users
relative-to=kerberos, digest-realm-name=ELYTRON.ORG},
groups-properties={path=kerberos-groups.properties, relative-to=kerberos})
```

Then define the Kerberos security factory for the server's identity.

```
./subsystem=elytron/kerberos-security-factory=test-server:add(relative-to=kerberos,
path=remote-test-server.keytab, principal=remote/test-server.elytron.org@ELYTRON.ORG)
```

Finally define the security domain and this time a SASL authentication factory.

```
./subsystem=elytron/security-domain=KerberosDomain:add(default-realm=kerberos-properties,
realms=[{realm=kerberos-properties, role-decoder=groups-to-roles}],
permission-mapper=default-permission-mapper)
./subsystem=elytron/sasl-authentication-factory=gssapi-authentication-factory:add(security-domain=
sasl-server-factory=elytron, mechanism-configurations=[{mechanism-name=GSSAPI,
credential-security-factory=test-server}])
```

This results in the following subsystem configuration: -

```
<subsystem xmlns="urn:wildfly:elytron:1.0" final-providers="combined-providers"
disallowed-providers="OracleUcrypto">
    ...
    <security-domains>
      ...
      <security-domain name="KerberosDomain" default-realm="kerberos-properties"
permission-mapper="default-permission-mapper">
        <realm name="kerberos-properties" role-decoder="groups-to-roles"/>
      </security-domain>
    </security-domains>
    <security-realms>
     ...
       <properties-realm name="kerberos-properties">
         <users-properties path="kerberos-users.properties" relative-to="kerberos"
digest-realm-name="ELYTRON.ORG"/>
         <groups-properties path="kerberos-groups.properties" relative-to="kerberos"/>
       </properties-realm>
     </security-realms>
     <credential-security-factories>
       <kerberos-security-factory name="test-server"
principal="remote/test-server.elytron.org@ELYTRON.ORG" path="remote-test-server.keytab"
relative-to="kerberos"/>
     </credential-security-factories>
     ...
     <sasl>
       ...
       <sasl-authentication-factory name="gssapi-authentication-factory"
sasl-server-factory="elytron" security-domain="KerberosDomain">
         <mechanism-configuration>
           <mechanism mechanism-name="GSSAPI" credential-security-factory="test-server"/>
         </mechanism-configuration>
       </sasl-authentication-factory>
       ...
     </sasl>
   </subsystem>
```

The management interface or Remoting connectors can now be updated to reference the SASL authentication factory.

The two Elytron examples defined here could also be combined into one to use a shared security domain and security realm and just use protocol specific authentication factories each referencing their own Kerberos security factory.

15.11.11 LDAP Authentication Migration

The section describing how to migrate from properties based authentication using either PicketBox or legacy security realms to Elytron also contained a lot of additional information regarding defining security domains, authentication factories, and how these are mapped to be used for authentication. This section will illustrate some equivalent LDAP configuration using legacy security realms and PicketBox security domains and show the equivalent configuration using Elytron but will not repeat the steps to wire it all together covered in the previous section.

These configuration examples are developed against a test LDAP sever with user entries like: -

```
dn: uid=TestUserOne,ou=users,dc=group-to-principal,dc=wildfly,dc=org
objectClass: top
objectClass: inetOrgPerson
objectClass: uidObject
objectClass: person
objectClass: organizationalPerson
cn: Test User One
sn: Test User One
uid: TestUserOne
userPassword: {SSHA}UG8ov2rnrnBKakcARVvraZHqTa7mFWJZlWt2HA==
```

The group entries then look like: -

```
dn: uid=GroupOne,ou=groups,dc=group-to-principal,dc=wildfly,dc=org
objectClass: top
objectClass: groupOfUniqueNames
objectClass: uidObject
cn: Group One
uid: GroupOne
uniqueMember: uid=TestUserOne,ou=users,dc=group-to-principal,dc=wildfly,dc=org
```

For authentication purposes the username will be matched against the 'uid' attribute, also the resulting group name will be taken from the 'uid' attribute of the group entry.

Legacy Security Realm

A connection to the LDAP server and related security realm can be created with the following commands: -

```
batch
./core-service=management/ldap-connection=MyLdapConnection:add(url="ldap://localhost:10389",
search-dn="uid=admin,ou=system", search-credential="secret")

./core-service=management/security-realm=LDAPRealm:add
./core-service=management/security-realm=LDAPRealm/authentication=ldap:add(connection="MyLdapConne
username-attribute=uid, base-dn="ou=users,dc=group-to-principal,dc=wildfly,dc=org")


./core-service=management/security-realm=LDAPRealm/authorization=ldap:add(connection=MyLdapConnect
base-dn="ou=users,dc=group-to-principal,dc=wildfly,dc=org")
./core-service=management/security-realm=LDAPRealm/authorization=ldap/group-search=group-to-princi
iterative=true, prefer-original-connection=true, principal-attribute=uniqueMember,
search-by=DISTINGUISHED_NAME, group-name=SIMPLE, group-name-attribute=uid)
run-batch
```

This results in the following configuration.

```
<management>
    <security-realms>
      ...
      <security-realm name="LDAPRealm">
        <authentication>
          <ldap connection="MyLdapConnection"
base-dn="ou=users,dc=group-to-principal,dc=wildfly,dc=org">
            <username-filter attribute="uid"/>
          </ldap>
        </authentication>
        <authorization>
          <ldap connection="MyLdapConnection">
            <username-to-dn>
              <username-filter base-dn="ou=users,dc=group-to-principal,dc=wildfly,dc=org"
attribute="uid"/>
            </username-to-dn>
            <group-search group-name="SIMPLE" iterative="true" group-name-attribute="uid">
              <group-to-principal search-by="DISTINGUISHED_NAME"
base-dn="ou=groups,dc=group-to-principal,dc=wildfly,dc=org" prefer-original-connection="true">
                <membership-filter principal-attribute="uniqueMember"/>
              </group-to-principal>
            </group-search>
          </ldap>
        </authorization>
      </security-realm>
    </security-realms>
    <outbound-connections>
      <ldap name="MyLdapConnection" url="ldap://localhost:10389" search-dn="uid=admin,ou=system"
search-credential="secret"/>
    </outbound-connections>
    ...
  </management>
```

PicketBox LdapExtLoginModule

The following commands can create a PicketBox security domain configured to use the LdapExtLoginModule to verify a username and password.

```
./subsystem=security/security-domain=application-security:add
./subsystem=security/security-domain=application-security/authentication=classic:add(login-modules=
flag=Required, module-options={ \
java.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory, \
java.naming.provider.url=ldap://localhost:10389, \
java.naming.security.authentication=simple, \
bindDN="uid=admin,ou=system", \
bindCredential=secret, \
baseCtxDN="ou=users,dc=group-to-principal,dc=wildfly,dc=org", \
baseFilter="(uid={0})", \
rolesCtxDN="ou=groups,dc=group-to-principal,dc=wildfly,dc=org",\
roleFilter="(uniqueMember={1})", \
roleAttributeID="uid" \
}])
```

This results in the following configuration.

```
<subsystem xmlns="urn:jboss:domain:security:2.0">
    ...
    <security-domains>
      ...
      <security-domain name="application-security">
        <authentication>
          <login-module code="LdapExtended" flag="required">
            <module-option name="java.naming.factory.initial"
value="com.sun.jndi.ldap.LdapCtxFactory"/>
            <module-option name="java.naming.provider.url" value="ldap://localhost:10389"/>
            <module-option name="java.naming.security.authentication" value="simple"/>
            <module-option name="bindDN" value="uid=admin,ou=system"/>
            <module-option name="bindCredential" value="secret"/>
            <module-option name="baseCtxDN"
value="ou=users,dc=group-to-principal,dc=wildfly,dc=org"/>
            <module-option name="baseFilter" value="(uid={0})"/>
            <module-option name="rolesCtxDN"
value="ou=groups,dc=group-to-principal,dc=wildfly,dc=org"/>
            <module-option name="roleFilter" value="(uniqueMember={1})"/>
            <module-option name="roleAttributeID" value="uid"/>
          </login-module>
        </authentication>
      </security-domain>
    </security-domains>
  </subsystem>
```

Migrated

Within the Elytron subsystem a directory context can be defined for the connection to LDAP: -

```
./subsystem=elytron/dir-context=ldap-connection:add(url=ldap://localhost:10389,
principal="uid=admin,ou=system", credential-reference={clear-text=secret})
```

Then a security realm can be created to search LDAP and verify the supplied password: -

```
./subsystem=elytron/ldap-realm=ldap-realm:add(dir-context=ldap-connection, \
direct-verification=true, \
identity-mapping={search-base-dn="ou=users,dc=group-to-principal,dc=wildfly,dc=org", \
rdn-identifier="uid", \
attribute-mapping=[{filter-base-dn="ou=groups,dc=group-to-principal,dc=wildfly,dc=org",filter="(un
```

In the prior two examples information is loaded from LDAP to use directly as groups or roles, in the Elytron case information can be loaded from LDAP to associate with the identity as attributes - these can subsequently be mapped to roles but attributes can be loaded for other purposes as well.

By default, if no `role-decoder` is defined for given `security-domain`, identity attribute "`Roles`" is mapped to the identity roles.

This leads to the following configuration.

```
<subsystem xmlns="urn:wildfly:elytron:1.0" final-providers="combined-providers"
disallowed-providers="OracleUcrypto">
    ...
    <security-realms>
      ...
      <ldap-realm name="ldap-realm" dir-context="ldap-connection" direct-verification="true">
        <identity-mapping rdn-identifier="uid"
search-base-dn="ou=users,dc=group-to-principal,dc=wildfly,dc=org">
          <attribute-mapping>
            <attribute from="uid" to="Roles" filter="(uniqueMember={1})"
filter-base-dn="ou=groups,dc=group-to-principal,dc=wildfly,dc=org"/>
          </attribute-mapping>
        </identity-mapping>
      </ldap-realm>
    </security-realms>
    ...
    <dir-contexts>
      <dir-context name="ldap-connection" url="ldap://localhost:10389"
principal="uid=admin,ou=system">
        <credential-reference clear-text="secret"/>
      </dir-context>
    </dir-contexts>
  </subsystem>
```

15.11.12 Properties Based Authentication / Authorization

PicketBox Based Configuration

This migration example assumes a deployed web application is configured to require authentication using FORM based authentication and is referencing a PicketBox based security domain using the UsersRolesLoginModule to load user information from a pair or properties files.

Original Configuration

A security domain can be defined in the legacy security subsystem using the following management operations: -

```
./subsystem=security/security-domain=application-security:add
./subsystem=security/security-domain=application-security/authentication=classic:add(login-modules=
flag=Required,
module-options={usersProperties=file://${jboss.server.config.dir}/example-users.properties,
rolesProperties=file://${jboss.server.config.dir}/example-roles.properties}}])
```

This would result in a security domain definition: -

```
<security-domain name="application-security">
    <authentication>
      <login-module code="UsersRoles" flag="required">
        <module-option name="usersProperties"
value="file://${jboss.server.config.dir}/example-users.properties"/>
        <module-option name="rolesProperties"
value="file://${jboss.server.config.dir}/example-roles.properties"/>
      </login-module>
    </authentication>
  </security-domain>
```

Intermediate Configuration

It is possible to take a previously defined PicketBox security domain and expose it as an Elytron security realm so it can be wired into a complete Elytron based configuration, if only properties based authentication was to be migrated it would be recommended to jump to the fully migration configuration and avoid the unnecessary dependency on the legacy security subsystem but for situations where that is not immediately possible these commands illustrate an intermediate solution.

These steps assume the original configuration is already in place.

The first step is to add a mapping to an Elytron security realm within the legacy security subsystem.

```
./subsystem=security/elytron-realm=application-security:add(legacy-jaas-config=application-securit
```

This results in the following configuration.

```
<subsystem xmlns="urn:jboss:domain:security:2.0">
    ...
    <elytron-integration>
      <security-realms>
        <elytron-realm name="application-security" legacy-jaas-config="application-security"/>
      </security-realms>
    </elytron-integration>
    ...
  </subsystem>
```

Within the Elytron subsystem a security domain can be defined which references the exported security realm and also a http authentication factory which supports FORM based authentication.

```
./subsystem=elytron/security-domain=application-security:add(realms=[{realm=application-security}]
default-realm=application-security, permission-mapper=default-permission-mapper)
./subsystem=elytron/http-authentication-factory=application-security-http:add(http-server-mechanis
security-domain=application-security, mechanism-configurations=[{mechanism-name=FORM}])
```

And the resulting configuration: -

```
<subsystem xmlns="urn:wildfly:elytron:1.0" final-providers="combined-providers"
disallowed-providers="OracleUcrypto">
    ...
    <security-domains>
      ...
      <security-domain name="application-security" default-realm="application-security"
permission-mapper="default-permission-mapper">
        <realm name="application-security"/>
      </security-domain>
    </security-domains>
    ...
    <http>
      ...
      <http-authentication-factory name="application-security-http"
http-server-mechanism-factory="global" security-domain="application-security">
        <mechanism-configuration>
          <mechanism mechanism-name="FORM"/>
        </mechanism-configuration>
      </http-authentication-factory>
      ...
    </http>
    ...
  </subsystem>
```

Finally configuration needs to be added to the Undertow subsystem to map the security domain referenced by the deployment to the newly defined http authentication factory.

```
./subsystem=undertow/application-security-domain=application-security:add(http-authentication-fact
```

Which results in: -

```
<subsystem xmlns="urn:jboss:domain:undertow:4.0">
    ...
    <application-security-domains>
      <application-security-domain name="application-security"
http-authentication-factory="application-security-http"/>
    </application-security-domains>
    ...
  </subsystem>
```

Note: If the deployment was already deployed at this point the application server should be reloaded or the deployment redeployed for the application security domain mapping to take effect.

The following command can then be used to verify the mapping was applied to the deployment.

```
[standalone@localhost:9990 /]
./subsystem=undertow/application-security-domain=application-security:read-resource(include-runtim
"outcome" => "success",
    "result" => {
        "enable-jacc" => false,
        "http-authentication-factory" => "application-security-http",
        "override-deployment-config" => false,
        "referencing-deployments" => ["HelloWorld.war"],
        "setting" => undefined
    }
}
```

The deployment being tested here is 'HelloWorld.war' and the output from the previous command shows this deployment is referencing the mapping.

At this stage the previously defined security domain is used for it's LoginModule configuration but this is wrapped by Elytron components which take over authentication.

Fully Migrated Configuration

Alternatively the configuration can be completely defined within the Elytron subsystem, in this case it is assumed none of the previous commands have been executed and this is started from a clean configuration - however if the security domain definition does exist in the legacy security subsystem that will remain completely independent.

First a new security realm can be defined within the Elytron subsystem referencing the files referenced previously: -

```
./subsystem=elytron/properties-realm=application-properties:add(users-properties={path=example-use
relative-to=jboss.server.config.dir, plain-text=true, digest-realm-name="Application Security"},
groups-properties={path=example-roles.properties, relative-to=jboss.server.config.dir},
groups-attribute=Roles)
```

As before a security domain and http authentication factory can be defined.

```
./subsystem=elytron/security-domain=application-security:add(realms=[{realm=application-properties
default-realm=application-properties, permission-mapper=default-permission-mapper)
./subsystem=elytron/http-authentication-factory=application-security-http:add(http-server-mechanis
security-domain=application-security, mechanism-configurations=[{mechanism-name=FORM}])
```

This results in the following overall configuration.

```
<subsystem xmlns="urn:wildfly:elytron:1.0" final-providers="combined-providers"
disallowed-providers="OracleUcrypto">
    ...
    <security-domains>
      ...
      <security-domain name="application-security" default-realm="application-properties"
permission-mapper="default-permission-mapper">
        <realm name="application-properties"/>
      </security-domain>
    </security-domains>
    <security-realms>
      ...
      <properties-realm name="application-properties" groups-attribute="Roles">
        <users-properties path="example-users.properties" relative-to="jboss.server.config.dir"
digest-realm-name="Application Security" plain-text="true"/>
        <groups-properties path="example-roles.properties"
relative-to="jboss.server.config.dir"/>
      </properties-realm>
    </security-realms>
    ...
    <http>
      ...
      <http-authentication-factory name="application-security-http"
http-server-mechanism-factory="global" security-domain="application-security">
        <mechanism-configuration>
          <mechanism mechanism-name="FORM"/>
        </mechanism-configuration>
      </http-authentication-factory>
      ...
    </http>
    ...
  </subsystem>
```

As before the application-security-domain mapping should be added to the Undertow subsystem and the server reloaded or the deployment redeployed as required.

```
./subsystem=undertow/application-security-domain=application-security:add(http-authentication-fact
```

Which results in: -

```
<subsystem xmlns="urn:jboss:domain:undertow:4.0">
    ...
    <application-security-domains>
      <application-security-domain name="application-security"
http-authentication-factory="application-security-http"/>
    </application-security-domains>
    ...
  </subsystem>
```

At this stage the authentication is the equivalent of the original configuration however now Elytron components are used exclusively.

Legacy Security Realm

Original Configuration

A legacy security realm can be defined using the following commands to load users passwords and group information from properties files.

```
./core-service=management/security-realm=ApplicationSecurity:add
./core-service=management/security-realm=ApplicationSecurity/authentication=properties:add(relativ
path=example-users.properties, plain-text=true)
./core-service=management/security-realm=ApplicationSecurity/authorization=properties:add(relative
path=example-roles.properties)
```

This results in the following realm definition.

```
<security-realm name="ApplicationSecurity">
    <authentication>
      <properties path="example-users.properties" relative-to="jboss.server.config.dir"
plain-text="true"/>
    </authentication>
    <authorization>
      <properties path="example-roles.properties" relative-to="jboss.server.config.dir"/>
    </authorization>
  </security-realm>
```

A legacy security realm would typically be used to secure either the management interfaces or remoting connectors.

Migrated Configuration

One of the motivations for adding the Elytron based security to the application server is to allow a consistent security solution to be used across the server, to replace the security realm the same steps as described in the previous 'Fully Migrated' section can be followed again up until the http-authentication-factory is defined.

A legacy security realm can also be used for SASL based authentication so a sasl-authentication-factory should also be defined.

```
./subsystem=elytron/sasl-authentication-factory=application-security-sasl:add(sasl-server-factory=
security-domain=application-security, mechanism-configurations=[{mechanism-name=PLAIN}])
```

```
<subsystem xmlns="urn:wildfly:elytron:1.0" final-providers="combined-providers"
disallowed-providers="OracleUcrypto">
    ...
    <sasl>
      ...
      <sasl-authentication-factory name="application-security-sasl"
sasl-server-factory="elytron" security-domain="application-security">
        <mechanism-configuration>
          <mechanism mechanism-name="PLAIN"/>
        </mechanism-configuration>
      </sasl-authentication-factory>
      ...
    </sasl>
  </subsystem>
```

This can be associated with a Remoting connector to use for authentication and the existing security realm reference cleared.

```
./subsystem=remoting/http-connector=http-remoting-connector:write-attribute(name=sasl-authenticati
value=application-security-sasl)
./subsystem=remoting/http-connector=http-remoting-connector:undefine-attribute(name=security-realm
```

```
<subsystem xmlns="urn:jboss:domain:remoting:4.0">
    ...
    <http-connector name="http-remoting-connector" connector-ref="default"
sasl-authentication-factory="application-security-sasl"/>
  </subsystem>
```

If this new configuration was to be used to secure the management interfaces more suitable names should be chosen but the following commands illustrate how to set the two authentication factories and clear the existing security realm reference.

```
./core-service=management/management-interface=http-interface:write-attribute(name=http-authentica
value=application-security-http)
./core-service=management/management-interface=http-interface:write-attribute(name=http-upgrade.sa
value=application-security-sasl)
./core-service=management/management-interface=http-interface:undefine-attribute(name=security-rea
```

```
<management-interfaces>
    <http-interface http-authentication-factory="application-security-http">
      <http-upgrade enabled="true" sasl-authentication-factory="application-security-sasl"/>
      <socket-binding http="management-http"/>
    </http-interface>
  </management-interfaces>
```

15.11.13 Security Properties

Lets suppose security properties "a" and "c" defined in legacy security:

```
<subsystem xmlns="urn:jboss:domain:security:2.0">
            ...
            <security-properties>
                <property name="a" value="b" />
                <property name="c" value="d" />
            </security-properties>
        </subsystem>
```

To define security properties in Elytron subsystem you need to set attribute `security-properties` of the subsystem:

```
./subsystem=elytron:write-attribute(name=security-properties, value={ \
    a = "b", \
    c = "d" \
})
```

You can also add or change one another property without modification of others using map operations. Following command will set property "e":

```
./subsystem=elytron:map-put(name=security-properties, key=e, value=f)
```

By the same way you can also remove one of properties - in example newly created property "e":

```
./subsystem=elytron:map-remove(name=security-properties, key=e)
```

Output XML configuration will be:

```
<subsystem xmlns="urn:wildfly:elytron:1.0" final-providers="combined-providers"
disallowed-providers="OracleUcrypto">
            <security-properties>
                <security-property name="a" value="b"/>
                <security-property name="c" value="d"/>
            </security-properties>
            ...
        </subsystem>
```

15.11.14 Security Vault Migration

Security Vault is primarily used in legacy configurations, a vault is used to store sensitive strings outside of the configuration files. WildFly server may only contain a single security vault.

Credential Store introduced in WildFly 11 is meant to expand Security Vault in terms of storing different credential types and introduce easy to implement SPI which allows to deploy custom implementations of CredentialStore SPI. Credentials are stored safely encrypted in storage file outside WildFly configuration files. Each WildFly server may contain multiple credential stores.

To easily migrate vault content into credential store we have added "vault" command into WildFly Elytron Tool. The tool could be found at $JBOSS_HOME/bin directory. It has several scripts named "elytron-tool.*" dependent on your platform of choice. One can use also simple form "java -jar $JBOSS_HOME/bin/wildfly-elytron-tool.jar <command> <arguments>" if it better suites ones needs.

Single Security Vault Conversion

To convert **single** security vault credential store use following example:

- to get sample vault use testing resources of Elytron Tool project from GitHub [1]

Command to run actual conversion:

```
./bin/elytron-tool.sh vault --enc-dir vault_data/ --keystore
vault-jceks.keystore --keystore-password MASK-2hKo56F1a3jYGnJwhPmiF5
--iteration 34 --salt 12345678 --alias test --location cs-v1.store --summary
```

Output:

```
Vault (enc-dir="vault_data/";keystore="vault-jceks.keystore") converted to
credential store "cs-v1.store"
Vault Conversion summary:
-------------------------------------
Vault Conversion Successful
CLI command to add new credential store:
/subsystem=elytron/credential-store=test:add(relative-to=jboss.server.data.dir,
```

Use elytron-tool.sh vault --help to get description of all parameters.

Notes:

- Elytron Tool cannot handle very first version of Security Vault data file.
- --keystore-password can come in two forms (1) masked as shown in the example or (2) clear text. Parameter --salt and --iteration are there to supply information to decrypt the masked password or to generate masked password in output. In case --salt and --iteration are omitted default values are used.
- When --summary parameter is specified, one can see nice output with CLI command to be used in WildFly console to add converted credential store to the configuration.

Bulk Security Vault Conversion

There is possibility to convert multiple vaults to credential store using --bulk-convert parameter with description file.
Example of description file from our tests [2]:

```
# Bulk conversion descriptor
keystore:target/test-classes/vault-v1/vault-jceks.keystore
keystore-password:MASK-2hKo56F1a3jYGnJwhPmiF5
enc-dir:target/test-classes/vault-v1/vault_data/
salt:12345678
iteration:34
location:target/v1-cs-1.store
alias:test

keystore:target/test-classes/vault-v1/vault-jceks.keystore
keystore-password:secretsecret
enc-dir:target/test-classes/vault-v1/vault_data/
location:target/v1-cs-2.store
alias:test

# different vault vault-v1-more
keystore:target/test-classes/vault-v1-more/vault-jceks.keystore
keystore-password:MASK-2hKo56F1a3jYGnJwhPmiF5
enc-dir:target/test-classes/vault-v1-more/vault_data/
salt:12345678
iteration:34
location:target/v1-cs-more.store
alias:test
```

After each "keystore:" option new conversion starts. All options are mandatory except "salt:", "iteration:" and "properties:"

Execute following command:

```
./bin/elytron-tool.sh vault --bulk-convert bulk-vault-conversion-desc
--summary
```

Output:

```
Vault
(enc-dir="vault-v1/vault_data/";keystore="vault-v1/vault-jceks.keystore")
converted to credential store "v1-cs-1.store"
Vault Conversion summary:
-------------------------------------
Vault Conversion Successful
CLI command to add new credential store:
/subsystem=elytron/credential-store=test:add(relative-to=jboss.server.data.dir,

-------------------------------------

Vault
(enc-dir="vault-v1/vault_data/";keystore="vault-v1/vault-jceks.keystore")
converted to credential store "v1-cs-2.store"
Vault Conversion summary:
-------------------------------------
Vault Conversion Successful
CLI command to add new credential store:
/subsystem=elytron/credential-store=test:add(relative-to=jboss.server.data.dir,

-------------------------------------

Vault
(enc-dir="vault-v1-more/vault_data/";keystore="vault-v1-more/vault-jceks.keysto
converted to credential store "v1-cs-more.store"
Vault Conversion summary:
-------------------------------------
Vault Conversion Successful
CLI command to add new credential store:
/subsystem=elytron/credential-store=test:add(relative-to=jboss.server.data.dir,

-------------------------------------
```

The result is conversion of all vaults with proper CLI commands.

References:

[1] https://github.com/wildfly-security/wildfly-elytron-tool/tree/master/src/test/resources/vault-v1
[2]
https://github.com/wildfly-security/wildfly-elytron-tool/blob/master/src/test/java/org/wildfly/security/tool/VaultCo

15.11.15 Simple SSL Migration

Simple SSL Migration

This section describe securing HTTP connections to the server using SSL using Elytron.
It suppose you have already configured SSL using legacy `security-realm`, for example by Admin Guide#Enable SSL, and your configuration looks like:

```
<security-realm name="ApplicationRealm">
  <server-identities>
    <ssl>
      <keystore path="server.keystore" relative-to="jboss.server.config.dir"
keystore-password="keystore_password" alias="server" key-password="key_password" />
    </ssl>
  </server-identities>
</security-realm>
```

To switch to Elytron you need to:

1. Create Elytron `key-store` - specifying where is the keystore file stored and password by which it is encrypted. Default type of keystore generated using keytool is JKS:

   ```
   /subsystem=elytron/key-store=LocalhostKeyStore:add(path=server.keystore,relative-to=jboss.se
   ```

2. Create Elytron `key-manager` - specifying keystore, alias (using `alias-filter`) and password of key:

   ```
   /subsystem=elytron/key-manager=LocalhostKeyManager:add(key-store=LocalhostKeyStore,alias-fil
   ```

3. Create Elytron `server-ssl-context` - specifying only reference to `key-manager` defined above:

   ```
   /subsystem=elytron/server-ssl-context=LocalhostSslContext:add(key-manager=LocalhostKeyManage
   ```

4. Switch `https-listener` from legacy `security-realm` to newly created Elytron `ssl-context`:

   ```
   /subsystem=undertow/server=default-server/https-listener=https:undefine-attribute(name=secur
   ```

5. And reload the server:

   ```
   reload
   ```

Output XML configuration of Elytron subsystem should look like:

```
<subsystem xmlns="urn:wildfly:elytron:1.0" ...>
            ...
            <tls>
                <key-stores>
                    <key-store name="LocalhostKeyStore">
                        <credential-reference clear-text="keystore_password"/>
                        <implementation type="JKS"/>
                        <file path="server.keystore" relative-to="jboss.server.config.dir"/>
                    </key-store>
                </key-stores>
                <key-managers>
                    <key-manager name="LocalhostKeyManager" key-store="LocalhostKeyStore">
                        <credential-reference clear-text="key_password"/>
                    </key-manager>
                </key-managers>
                <server-ssl-contexts>
                    <server-ssl-context name="LocalhostSslContext"
key-manager="LocalhostKeyManager"/>
                </server-ssl-contexts>
            </tls>
        </subsystem>
```

Output `https-listener` in Undertow subsystem should be:

```
<https-listener name="https" socket-binding="https" ssl-context="LocalhostSslContext"
enable-http2="true"/>
```

Client-Cert SSL Authentication Migration

This suppose you have already configured Client-Cert SSL authentication using `truststore` in legacy `security-realm`, for example by Admin Guide#Add Client-Cert to SSL, and your configuration looks like:

```
<security-realm name="ApplicationRealm">
  <server-identities>
    <ssl>
      <keystore path="server.keystore" relative-to="jboss.server.config.dir"
keystore-password="keystore_password" alias="server" key-password="key_password" />
    </ssl>
  </server-identities>
  <authentication>
    <truststore path="server.truststore" relative-to="jboss.server.config.dir"
keystore-password="truststore_password" />
    <local default-user="$local"/>
    <properties path="application-users.properties" relative-to="jboss.server.config.dir"/>
  </authentication>
</security-realm>
```

Following configuration is sufficient to prevent users without valid certificate and private key to access the server, but it does not provide user identity to the application. That require to define `CLIENT_CERT` HTTP mechanism / `EXTERNAL` SASL mechanism, which will be described later.)

At first use steps above to migrate basic part of the configuration. Then continue by following:

1. Create `key-store` of truststore - like for keystore above:

```
/subsystem=elytron/key-store=TrustStore:add(path=server.truststore,relative-to=jboss.server.
```

2. Create `trust-manager` - specifying `key-store` of trustore, created above:

```
/subsystem=elytron/trust-manager=TrustManager:add(key-store=TrustStore)
```

3. Modify `server-ssl-context` to use newly created trustmanager:

```
/subsystem=elytron/server-ssl-context=LocalhostSslContext:write-attribute(name=trust-manager
```

4. Enable client authentication for `server-ssl-context`:

```
/subsystem=elytron/server-ssl-context=LocalhostSslContext:write-attribute(name=need-client-a
```

5. And reload the server:

```
reload
```

Output XML configuration of Elytron subsystem should look like:

```
<subsystem xmlns="urn:wildfly:elytron:1.0" ...>
            ...
            <tls>
                <key-stores>
                    <key-store name="LocalhostKeyStore">
                        <credential-reference clear-text="keystore_password"/>
                        <implementation type="JKS"/>
                        <file path="server.keystore" relative-to="jboss.server.config.dir"/>
                    </key-store>
                    <key-store name="TrustStore">
                        <credential-reference clear-text="truststore_password"/>
                        <implementation type="JKS"/>
                        <file path="server.truststore" relative-to="jboss.server.config.dir"/>
                    </key-store>
                </key-stores>
                <key-managers>
                    <key-manager name="LocalhostKeyManager" key-store="LocalhostKeyStore"
alias-filter="server">
                        <credential-reference clear-text="key_password"/>
                    </key-manager>
                </key-managers>
                <trust-managers>
                    <trust-manager name="TrustManager" key-store="TrustStore"/>
                </trust-managers>
                <server-ssl-contexts>
                    <server-ssl-context name="LocalhostSslContext" need-client-auth="true"
key-manager="LocalhostKeyManager" trust-manager="TrustManager"/>
                </server-ssl-contexts>
            </tls>
        </subsystem>
```

15.11.16 SSL with Client Cert Migration

As this documentation is primarily intended for users migrating to WildFly Elytron I am going to jump straight into the configuration required with WildFly Elytron.

This section will cover how to create the various resources required to achieve CLIENT_CERT authentication with fallback to username / password authentication for both HTTP and SASL (i.e. Remoting) - both are being covered at the same time as predominantly they require the same core configuration, it is not until the definition of the authentication factories that the configuration becomes really specific.

The WildFly Elytron project already contains a dummy certificate authority set up that we use for testing, the KeyStores within this documentation are from the dummy certificate authority although for anything other than testing these should be replaced with your own.

As a first step we define some paths that point to the wildfly-elytron project so we can use these in the subsequent configuration.

```
./path=elytron.project:add(path=/home/darranl/src/wildfly10/wildfly-elytron)
./path=elytron.project.jks:add(path=src/test/resources/ca/jks, relative-to=elytron.project)
./path=elytron.project.properties:add(path=src/test/resources/org/wildfly/security/auth/realm,
relative-to=elytron.project)
```

This results in the following configuration.

```
<paths>
    <path name="elytron.project" path="/home/darranl/src/wildfly10/wildfly-elytron"/>
    <path name="elytron.project.jks" path="src/test/resources/ca/jks"
relative-to="elytron.project"/>
    <path name="elytron.project.properties"
path="src/test/resources/org/wildfly/security/auth/realm" relative-to="elytron.project"/>
  </paths>
```

KeyStores, KeyManagers, and TrustManagers.

The next step is to define the KeyStore resources, for this example three different keystores are used: -

1. **localhost.keystore** - Contains the servers key and certificate for 'localhost'.
2. **beetles.keystore** - Contains the individual client certificates.
3. **ca.keystore** - Contains the certificate of the certificate authority.

When we define the overall configuration we will use the localhost keystore along with the ca keystore for the incoming connections so initially all client certificates signed by the certificate authority will be accepted and subsequently a security realm will check it against the actual certificates within beetles keystore.

```
./subsystem=elytron/key-store=localhost:add(type=jks, relative-to=elytron.project.jks,
path=localhost.keystore, credential-reference={clear-text=Elytron})
./subsystem=elytron/key-store=beetles:add(type=jks, relative-to=elytron.project.jks,
path=beetles.keystore, credential-reference={clear-text=Elytron})
./subsystem=elytron/key-store=ca:add(type=jks, relative-to=elytron.project.jks,
path=ca.truststore, credential-reference={clear-text=Elytron})
```

This results in the following definitions: -

```
<subsystem xmlns="urn:wildfly:elytron:1.1" final-providers="combined-providers"
disallowed-providers="OracleUcrypto">
    ...
    <tls>
      <key-stores>
        <key-store name="localhost">
          <credential-reference clear-text="Elytron"/>
          <implementation type="jks"/>
          <file path="localhost.keystore" relative-to="elytron.project.jks"/>
        </key-store>
        <key-store name="beetles">
          <credential-reference clear-text="Elytron"/>
          <implementation type="jks"/>
          <file path="beetles.keystore" relative-to="elytron.project.jks"/>
        </key-store>
        <key-store name="ca">
          <credential-reference clear-text="Elytron"/>
          <implementation type="jks"/>
          <file path="ca.truststore" relative-to="elytron.project.jks"/>
        </key-store>
      </key-stores>
    </tls>
  </subsystem>
```

Next the key and trust manager resources will be defined using these keystores.

```
./subsystem=elytron/key-manager=localhost-manager:add(algorithm=SunX509, key-store=localhost,
credential-reference={clear-text=Elytron})
./subsystem=elytron/trust-manager=ca-manager:add(algorithm=SunX509, key-store=ca)
```

Resulting in: -

```
<subsystem xmlns="urn:wildfly:elytron:1.1" final-providers="combined-providers"
disallowed-providers="OracleUcrypto">
    ...
    <tls>
     ...
       <key-managers>
         <key-manager name="localhost-manager" algorithm="SunX509" key-store="localhost">
           <credential-reference clear-text="Elytron"/>
         </key-manager>
       </key-managers>
       <trust-managers>
         <trust-manager name="ca-manager" algorithm="SunX509" key-store="ca"/>
       </trust-managers>
    </tls>
  </subsystem>
```

Realms and Domains

Two security realms are now defined, one of these uses properties files from within the WildFly Elytron project to support username/password authentication and the other using the clients certificates for verification.

```
./subsystem=elytron/properties-realm=test-users:add(users-properties={relative-to=elytron.project.
path=clear.properties, plain-text=true, digest-realm-name=ManagementRealm},
groups-properties={relative-to=elytron.project.properties, path=groups.properties})
./subsystem=elytron/key-store-realm=key-store-realm:add(key-store=beetles)
```

Resulting in: -

```
<subsystem xmlns="urn:wildfly:elytron:1.1" final-providers="combined-providers"
disallowed-providers="OracleUcrypto">
    ...
    <security-realms>
      ...
      <key-store-realm name="key-store-realm" key-store="beetles"/>
      ...
      <properties-realm name="test-users">
        <users-properties path="clear.properties" relative-to="elytron.project.properties"
digest-realm-name="ManagementRealm" plain-text="true"/>
        <groups-properties path="groups.properties" relative-to="elytron.project.properties"/>
      </properties-realm>
    </security-realms>
    ...
  </subsystem>
```

These security realms can now be referenced from a security domain: -

```
./subsystem=elytron/security-domain=client-cert-domain:add(realms=[{realm=test-users},{realm=key-s
\
  default-realm=test-users, \
  permission-mapper=default-permission-mapper)
```

Resulting in: -

```
<subsystem xmlns="urn:wildfly:elytron:1.1" final-providers="combined-providers"
disallowed-providers="OracleUcrypto">
    ...
    <security-domains>
      ...
      <security-domain name="client-cert-domain" default-realm="test-users"
permission-mapper="default-permission-mapper">
        <realm name="test-users"/>
        <realm name="key-store-realm"/>
      </security-domain>
    </security-domains>
    ...
  </subsystem>
```

Before moving onto the individual authentication factories a couple of additional utility resources are also required: -

```
./subsystem=elytron/constant-realm-mapper=key-store-realm:add(realm-name=key-store-realm)
./subsystem=elytron/x500-attribute-principal-decoder=x500-decoder:add(attribute-name=CN,
maximum-segments=1)
```

Resulting in: -

```
<subsystem xmlns="urn:wildfly:elytron:1.1" final-providers="combined-providers"
disallowed-providers="OracleUcrypto">
    ...
    <mappers>
      ...
      <x500-attribute-principal-decoder name="x500-decoder" attribute-name="CN"
maximum-segments="1"/>
      ...
      <constant-realm-mapper name="key-store-realm" realm-name="key-store-realm"/>
      ...
    </mappers>
    ...
  </subsystem>
```

HTTP Authentication Factory

For the HTTP connections we now define a HTTP authentication factory using the previously defined resources and it is configured to support CLIENT_CERT and DIGEST authentication.

```
./subsystem=elytron/http-authentication-factory=client-cert-digest:add(http-server-mechanism-facto
\
  security-domain=client-cert-domain, \
 mechanism-configurations=[{ \
  mechanism-name=CLIENT_CERT, \
  realm-mapper=key-store-realm, \
  pre-realm-principal-transformer=x500-decoder}, \
 {mechanism-name=DIGEST, mechanism-realm-configurations=[{realm-name=ManagementRealm}]}])
```

Resulting in: -

```
<subsystem xmlns="urn:wildfly:elytron:1.1" final-providers="combined-providers"
disallowed-providers="OracleUcrypto">
    ...
    <http>
      ...
      <http-authentication-factory name="client-cert-digest"
http-server-mechanism-factory="global" security-domain="client-cert-domain">
        <mechanism-configuration>
          <mechanism mechanism-name="CLIENT_CERT" pre-realm-principal-transformer="x500-decoder"
realm-mapper="key-store-realm"/>
          <mechanism mechanism-name="DIGEST">
            <mechanism-realm realm-name="ManagementRealm"/>
          </mechanism>
        </mechanism-configuration>
      </http-authentication-factory>
      ...
    </http>
    ...
  </subsystem>
```

Where DIGEST authentication is used we rely on the default configuration within the security domain to select the 'test-users' realm, however where CLIENT_CERT authentication is in use an alternative realm-mapper is referenced to ensure the 'key-store-realm' is used.

Additionally for CLIENT_CERT authentication a principal-transformer is referenced to extract the CN attribute from the distinguished name of the client certificate and use this when accessing the identity from the security realm.

SASL Authentication Factory

The architecture of the two authentication factories if very similar so a SASL authentication factory can be defined in the same way as the HTTP equivalent.

```
./subsystem=elytron/sasl-authentication-factory=client-cert-digest:add(sasl-server-factory=elytron
\
  security-domain=client-cert-domain, \
  mechanism-configurations=[{mechanism-name=EXTERNAL, \
  realm-mapper=key-store-realm, \
  pre-realm-principal-transformer=x500-decoder}, \
  {mechanism-name=DIGEST-MD5, mechanism-realm-configurations=[{realm-name=ManagementRealm}]}])
```

This results in: -

```
<subsystem xmlns="urn:wildfly:elytron:1.1" final-providers="combined-providers"
disallowed-providers="OracleUcrypto">
    ...
    <sasl>
      ...
      <sasl-authentication-factory name="client-cert-digest" sasl-server-factory="elytron"
security-domain="client-cert-domain">
        <mechanism-configuration>
          <mechanism mechanism-name="EXTERNAL" pre-realm-principal-transformer="x500-decoder"
realm-mapper="key-store-realm"/>
          <mechanism mechanism-name="DIGEST-MD5">
            <mechanism-realm realm-name="ManagementRealm"/>
          </mechanism>
        </mechanism-configuration>
      </sasl-authentication-factory>
      ...
    </sasl>
    ...
  </subsystem>
```

Realm mappers and principal transformers are defined in the same way as were defined for HTTP.

SSL Context

An SSL context is also defined for use by the server.

```
./subsystem=elytron/server-ssl-context=localhost:add(key-manager=localhost-manager,
trust-manager=ca-manager, \
  security-domain=client-cert-domain, \
  authentication-optional=true, \
  want-client-auth=true, \
  need-client-auth=false)
```

Resulting in: -

```
<subsystem xmlns="urn:wildfly:elytron:1.1" final-providers="combined-providers"
disallowed-providers="OracleUcrypto">
    ...
    <tls>
      ...
      <server-ssl-contexts>
        <server-ssl-context name="localhost" security-domain="client-cert-domain"
want-client-auth="true" need-client-auth="false" authentication-optional="true"
key-manager="localhost-manager" trust-manager="ca-manager"/>
      </server-ssl-contexts>
    </tls>
  </subsystem>
```

As we will be supporting fallback to username/password authentication *need-client-auth* is set to *false* as well as *authentication-optional* being set to *false*, this allows connections to be established but an alternative form of authentication will be required.

Using for Management

At this point the management interfaces can be updated to use the newly defined resources, we need to add references to the two new authentication factories and the SSL context, we can also remove the existing reference to the legacy security realm. As this is modifying existing interfaces a server reload will also be required.

```
./core-service=management/management-interface=http-interface:write-attribute(name=ssl-context,
value=localhost)
./core-service=management/management-interface=http-interface:write-attribute(name=secure-socket-b
value=management-https)
./core-service=management/management-interface=http-interface:write-attribute(name=http-authentica
value=client-cert-digest)
./core-service=management/management-interface=http-interface:write-attribute(name=http-upgrade.sa
value=client-cert-digest)
./core-service=management/management-interface=http-interface:undefine-attribute(name=security-rea
```

The management interface configuration then becomes: -

```
<management>
    ...
    <management-interfaces>
      <http-interface http-authentication-factory="client-cert-digest" ssl-context="localhost">
        <http-upgrade enabled="true" sasl-authentication-factory="client-cert-digest"/>
        <socket-binding http="management-http" https="management-https"/>
      </http-interface>
    </management-interfaces>
    ...
  </management>
```

Admin Clients

At this stage assuming the same files have been used as in this example it should be possible to connect to the management interface of the server either using a web browser or the JBoss CLI with the username *elytron* and password *passwd12#$*

For certificate based authentication the keys and certificates from the WildFly Elytron tests can be used, these are found in JKS keystores under 'src/test/resources/ca/jks', these keystores have a password of *Elytron*.

Web Browser Configuration

A PKCS#12 file can be created from the test keystores,this can then be imported into the web browser to use when connecting to the server.

```
keytool -importkeystore -srckeystore ladybird.keystore \
  -destkeystore ladybird.pkcs12 \
  -srcstoretype jks \
  -deststoretype pkcs12 \
  -deststorepass Elytron \
  -srcalias ladybird \
  -destalias ladybird
```

CLI Configuration

Since the integration of WildFly Elytron it is possible with the CLI to use a configuration file wildfly-config.xml to define the security settings including the settings for the client side SSL context.

For the purpose of this example copy the ladybird,keystore and ca.truststore from the Wildfly Elytron testsuite to the location the JBoss CLI is being started from, the following wildfly-config.xml can be created in this location as well: -

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration>
    <authentication-client xmlns="urn:elytron:1.0">
        <key-stores>
            <key-store name="ladybird" type="jks" >
                <file name="ladybird.keystore"/>
                <key-store-clear-password password="Elytron" />
            </key-store>
            <key-store name="ca" type="jks">
                <file name="ca.truststore"/>
                <key-store-clear-password password="Elytron" />
            </key-store>
        </key-stores>
        <ssl-context-rules>
            <rule use-ssl-context="default" />
        </ssl-context-rules>
        <ssl-contexts>
            <ssl-context name="default">
                <key-store-ssl-certificate key-store-name="ladybird" alias="ladybird">
                    <key-store-clear-password password="Elytron" />
                </key-store-ssl-certificate>
                <trust-store key-store-name="ca" />
            </ssl-context>
        </ssl-contexts>
    </authentication-client>
</configuration>
```

The CLI can now be started using the following command: -

```
./jboss-cli.sh -c -Dwildfly.config.url=wildfly-config.xml
```

The *:whoami* command can be used within the CLI to double check the current identity.

```
[standalone@localhost:9993 /] :whoami(verbose=true)
{
    "outcome" => "success",
    "result" => {
        "identity" => {"username" => "Ladybird"},
        "mapped-roles" => ["SuperUser"]
    }
}
```

15.12 OpenSSL

15.13 Protecting Wildfly Adminstration Console With Keycloak

15.13.1 Overview

In this document you will learn how to integrate security for Wildfly Administration Console with Keycloak using Elytron subsystem.

15.13.2 System Requirements

To follow the instructions in this document, make sure you have both Wildfly and Keycloak servers properly installed. You need the latest versions for both servers.

When running Wildfly, it must be using port 8080 (default port). The following command can be used to start the server:

```
WILDFLY_HOME/bin/standalone.sh
```

For Keycloak, use the following command to start the server on port 8180:

```
KEYCLOAK_HOME/bin/standalone.sh \-Djboss.socket.binding.port-offset=100
```

15.13.3 Installing Keycloak Wildfly Elytron Adapters

Keycloak integration is only possible when using Keycloak Wildfly Elytron Adapter. This adapter is fully integrated with the new security infrastructure in Wildfly provided by Elytron and its subsystem.

Download the latest version of Wildfly Client Adapters and follow the instructions in this document to extract/install the adapters in your Wildfly installation. Make sure you run the following script when installing the adapter:

```
WILDFLY_HOME/bin/jboss-cli.sh --file=adapter-elytron-install-offline.cli
```

15.13.4 Creating a Keycloak Realm for Wildfly Management Services

We'll be protecting both administration console and HTTP management interface in Wildfly. For that, we need to create a Keycloak realm and two client applications, where these clients will be used to configure security for both administration console and HTTP management interface.

Start your Keycloak server using the following command:

```
KEYCLOAK_HOME/bin/standalone.sh -Djboss.socket.binding.port-offset=100
```

After running the command above you should be able to access Keycloak Administration Console at http://localhost:8180/auth and log in.

If you are running the server for the first name, you will be prompted to create an initial admin user to get started. Once you provide the username and password for the admin user you'll be redirected to Keycloak Administration Console login page.

Create a realm with a name **wildfly-infra**.

Create a client application with a name **wildfly-console** and configure it as follows:

- Select **public** in the **Access Type** field
- Add a new **Valid Redirect URI** with a value http://localhost:9990/console/*
- Add a new **Web Origins** with a value http://localhost:9990

Save changes for client wildfly-console and make sure it is properly updated. The client should have a configuration similar to following:

Create another client application with a name **wildfly-management** and configure it as follows:

- Select **bearer-only** in the **Access Type** field

Save changes for client wildfly-management and make sure it is properly updated. The client should have a configuration similar to following:

For last, we need to create an user to **jboss-infra** realm and also a role to grant to this user access to the Wildfly Administration Console.

Create a **Realm Role** with a name **ADMINISTRATOR**. It is important to keep the name in uppercase.

For example purposes, we are only using the ADMINISTRATOR role to grant users access to the administration console. However, Wildfly also supports other roles with different access scopes. For more details, please take a look at https://docs.jboss.org/author/display/WFLY/RBAC.

Create a new user with a name **admin**. You can choose whatever password you like, just make sure you set one. After creating the user, map the **ADMINISTRATOR** role to the **admin** user.

15.13.5 Protecting Wildfly Console and Management API

As a last configuration step, you need to configure Keycloak, Elytron and core subsystems to protect both management services.

Copy and paste the following commands to a new file with a name **protect-wildfly-mgmt-services.cli:**

```
# Create a realm for both wildfly console and mgmt interface
/subsystem=keycloak/realm=jboss-infra:add(auth-server-url=http://localhost:8180/auth,realm-public-
Create a secure-deployment in order to protect mgmt interface
/subsystem=keycloak/secure-deployment=wildfly-management:add(realm=jboss-infra,resource=wildfly-ma
Protect HTTP mgmt interface with Keycloak adapter
/core-service=management/management-interface=http-interface:undefine-attribute(name=security-realm
value={enabled=true, sasl-authentication-factory=management-sasl-authentication})

# Enable RBAC where roles are obtained from the identity
/core-service=management/access=authorization:write-attribute(name=provider,value=rbac)
/core-service=management/access=authorization:write-attribute(name=use-identity-roles,value=true)
Create a secure-server in order to publish the wildfly console configuration via mgmt interface
/subsystem=keycloak/secure-server=wildfly-console:add(realm=jboss-infra,resource=wildfly-console,p
reload
reload
```

Before saving the new file, you need to obtain the public key of **jboss-infra** realm and replace **[REALM_PUBLIC_KEY]** in the first command above with the value of the public key. To obtain realm's public key, go to Keycloak Administration Console, select **Realm Settings** on the left side menu and than click on the **Keys** tab. You should see a page as follows:

For last, execute the **protect-wildfly-mgmt-services.cli** script using JBoss CLI. Make sure your Wildfly instance is running before running the script:

```
WILFLY_HOME/bin/jboss-cli.sh --connect --file=protect-wildfly-mgmt-services.cli
```

15.13.6 Accessing Wildfly Administration Console

If everything is correct you should be able to access the Wildfly Administration Console now after authenticating in Keycloak.

Try to access Wildfly Administration Console and you should be redirected to a login page in Keycloak. You should be able to log in as the **admin** user you created in the **jboss-infra** realm.

15.14 Using the Elytron Subsystem

15.14.1 Set Up and Configure Authentication for Applications

Configure Authentication with a Properties File-Based Identity Store

Create properties files:

You need to create two properties files: one that maps user to passwords and another that maps users to roles. Usually these files are located in the *jboss.server.config.dir* directory and follow the naming convention **-users.properties* and **-roles.properties*, but other locations and names may be used. The **-users.properties* file must also contain a reference to the *properties-realm*, which you will create in the next step: *#$REALM_NAME=YOUR_PROPERTIES_REALM_NAME$*

Example user to password file: example-users.properties

```
#$REALM_NAME=examplePropRealm$
user1=password123
user2=password123
```

Example user to roles file: example-roles.properties

```
user1=Admin
user2=Guest
```

Configure a properties-realm in WildFly:

```
/subsystem=elytron/properties-realm=examplePropRealm:add(groups-attribute=groups,groups-properties
```

The name of the *properties-realm* is *examplePropRealm*, which is used in the previous step in the *example-users.properties* file. Also, if your properties files are located outside of *jboss.server.config.dir*, then you need to change the *path* and *relative-to* values appropriately.

Configure a security-domain :

```
/subsystem=elytron/security-domain=exampleSD:add(realms=[{realm=examplePropRealm,role-decoder=grou
```

Configure an http-authentication-factory :

```
/subsystem=elytron/http-authentication-factory=example-http-auth:add(http-server-mechanism-factory
```

This example shows creating an *http-authentication-factory* using *BASIC* authentication, but it could be updated to other mechanisms such as *FORM*.

Configure an application-security-domain in the Undertow subsystem:

```
/subsystem=undertow/application-security-domain=exampleApplicationDomain:add(http-authentication-f
```

Configure your application's web.xml and jboss-web.xml .

Your application's *web.xml* and *jboss-web.xml* must be updated to use the *application-security-domain* you configured in WildFly. An example of this is available in the Configure Applications to Use Elytron or Legacy Security for Authentication section.

Configure Authentication with a Filesystem-Based Identity Store

Chose a directory for users:

You need a directory where your users will be stored. In this example, we are using a directory called *fs-realm-users* located in *jboss.server.config.dir*.

Configure a filesystem-realm in WildFly:

```
/subsystem=elytron/filesystem-realm=exampleFsRealm:add(path=fs-realm-users,relative-to=jboss.serve
```

If your directory is located outside of *jboss.server.config.dir*, then you need to change the *path* and *relative-to* values appropriately.

Add a user:

When using the *filesystem-realm*, you can add users using the management CLI.

```
/subsystem=elytron/filesystem-realm=exampleFsRealm/identity=user1:add()
/subsystem=elytron/filesystem-realm=exampleFsRealm/identity=user1:set-password(
clear={password="password123"})
/subsystem=elytron/filesystem-realm=exampleFsRealm/identity=user1:add-attribute(name=Roles,
value=["Admin","Guest"])
```

Add a simple-role-decoder :

```
/subsystem=elytron/simple-role-decoder=from-roles-attribute:add(attribute=Roles)
```

This *simple-role-decoder* decodes a principal's roles from the *Roles* attribute. You can change this value if your roles are in a different attribute.

Configure a security-domain :

```
/subsystem=elytron/security-domain=exampleFsSD:add(realms=[{realm=exampleFsRealm,role-decoder=from
```

Configure an http-authentication-factory :

```
/subsystem=elytron/http-authentication-factory=example-fs-http-auth:add(http-server-mechanism-fact
```

This example shows creating an *http-authentication-factory* using *BASIC* authentication, but it could be updated to other mechanisms such as *FORM*.

Configure an application-security-domain in the Undertow subsystem:

```
/subsystem=undertow/application-security-domain=exampleApplicationDomain:add(http-authentication-f
```

Configure your application's web.xml and jboss-web.xml .

Your application's *web.xml* and *jboss-web.xml* must be updated to use the *application-security-domain* you configured in WildFly. An example of this is available in the Configure Applications to Use Elytron or Legacy Security for Authentication section.

Your application is now using a filesystem-based identity store for authentication.

Configure Authentication with a Database Identity Store

Determine your database format for usernames, passwords, and roles:

To set up authentication using a database for an identity store, you need to determine how your usernames, passwords, and roles are stored in that database. In this example, we are using a single table with the following sample data:

username	password	roles
user1	password123	Admin
user2	password123	Guest

Configure a datasource:

To connect to a database from WildFly, you must have the appropriate database driver deployed as well as a datasource configured. This example shows deploying the driver for postgres and configuring a datasource in WildFly:

```
deploy /path/to/postgresql-9.4.1210.jar

data-source add --name=examplePostgresDS --jndi-name=java:jboss/examplePostgresDS
--driver-name=postgresql-9.4.1210.jar
--connection-url=jdbc:postgresql://localhost:5432/postgresdb --user-name=postgresAdmin
--password=mysecretpassword
```

Configure a jdbc-realm in WildFly:

```
/subsystem=elytron/jdbc-realm=exampleDbRealm:add(principal-query=[{sql="SELECT password,roles
FROM wildfly_users WHERE
username=?",data-source=examplePostgresDS,clear-password-mapper={password-index=1},attribute-mappi
```

NOTE: The above example shows how to obtain passwords and roles from a single *principal-query*. You can also create additional *principal-query* with *attribute-mapping* attributes if you require multiple queries to obtain roles or additional authentication or authorization information.

Configure a security-domain :

```
/subsystem=elytron/security-domain=exampleDbSD:add(realms=[{realm=exampleDbRealm,role-decoder=grou
```

Configure an http-authentication-factory :

```
/subsystem=elytron/http-authentication-factory=example-db-http-auth:add(http-server-mechanism-fact
```

This example shows creating an *http-authentication-factory* using *BASIC* authentication, but it could be updated to other mechanisms such as *FORM*.

Configure an application-security-domain in the Undertow subsystem:

```
/subsystem=undertow/application-security-domain=exampleApplicationDomain:add(http-authentication-f
```

Configure your application's web.xml and jboss-web.xml .

Your application's *web.xml* and *jboss-web.xml* must be updated to use the *application-security-domain* you configured in WildFly. An example of this is available in the Configure Applications to Use Elytron or Legacy Security for Authentication section.

Configure Authentication with an LDAP-Based Identity Store

Determine your LDAP format for usernames, passwords, and roles:

To set up authentication using an LDAP server for an identity store, you need to determine how your usernames, passwords, and roles are stored. In this example, we are using the following structure:

```
dn: dc=wildfly,dc=org
dc: wildfly
objectClass: top
objectClass: domain

dn: ou=Users,dc=wildfly,dc=org
objectClass: organizationalUnit
objectClass: top
ou: Users

dn: uid=jsmith,ou=Users,dc=wildfly,dc=org
objectClass: top
objectClass: person
objectClass: inetOrgPerson
cn: John Smith
sn: smith
uid: jsmith
userPassword: password123

dn: ou=Roles,dc=wildfly,dc=org
objectclass: top
objectclass: organizationalUnit
ou: Roles

dn: cn=Admin,ou=Roles,dc=wildfly,dc=org
objectClass: top
objectClass: groupOfNames
cn: Admin
member: uid=jsmith,ou=Users,dc=wildfly,dc=org
```

Configure a dir-context :

To connect to the LDAP server from WildFly, you need to configure a *dir-context* that provides the URL as well as the principal used to connect to the server.

```
/subsystem=elytron/dir-context=exampleDC:add(url="ldap://127.0.0.1:10389",principal="uid=admin,ou=
```

Configure an ldap-realm in WildFly:

```
/subsystem=elytron/ldap-realm=exampleLR:add(dir-context=exampleDC,identity-mapping={search-base-dn
```

Add a simple-role-decoder :

```
/subsystem=elytron/simple-role-decoder=from-roles-attribute:add(attribute=Roles)
```

Configure a security-domain :

```
/subsystem=elytron/security-domain=exampleLdapSD:add(realms=[{realm=exampleLR,role-decoder=from-ro
```

Configure an http-authentication-factory :

```
/subsystem=elytron/http-authentication-factory=example-ldap-http-auth:add(http-server-mechanism-fa
```

This example shows creating an *http-authentication-factory* using *BASIC* authentication, but it could be updated to other mechanisms such as *FORM*.

Configure an application-security-domain in the Undertow subsystem:

```
/subsystem=undertow/application-security-domain=exampleApplicationDomain:add(http-authentication-f
```

Configure your application's web.xml and jboss-web.xml .

Your application's *web.xml* and *jboss-web.xml* must be updated to use the *application-security-domain* you configured in WildFly. An example of this is available in the Configure Applications to Use Elytron or Legacy Security for Authentication section.

IMPORTANT: In cases where you configure an LDAP server in the *elytron* subsystem for authentication and that LDAP server then becomes unreachable, WildFly will return a *500*, or internal server error, error code when attempting authentication using that unreachable LDAP server. This behavior differs from the legacy *security* subsystem, which will return a *401*, or unauthorized, error code under the same conditions.

Configure Authentication with Certificates

IMPORTANT: Before you can set up certificate-based authentication, you must have two-way SSL configured.

Configure a key-store-realm .

```
/subsystem=elytron/key-store-realm=ksRealm:add(key-store=twoWayTS)
```

You must configure this realm with a truststore that contains the client's certificate. The authentication process uses the same certificate presented by the client during the two-way SSL handshake.

Create a Decoder.

You need to create a *x500-attribute-principal-decoder* to decode the principal you get from your certificate. The below example will decode the principal based on the first *CN* value.

```
/subsystem=elytron/x500-attribute-principal-decoder=CNDecoder:add(oid="2.5.4.3",maximum-segments=1
```

For example, if the full *DN* was *CN=client,CN=client-certificate,DC=example,DC=jboss,DC=org*, *CNDecoder* would decode the principal as *client*. This decoded principal is used as the *alias* value to lookup a certificate in the truststore configured in *ksRealm*.

IMPORTANT: The decoded principal *__MUST__* must be the *alias* value you set in your server's truststore for the client's certificate.

Add a constant-role-mapper for assigning roles.

This is example uses a *constant-role-mapper* to assign roles to a principal from *ksRealm* but other approaches may also be used.

```
/subsystem=elytron/constant-role-mapper=constantClientCertRole:add(roles=[Admin,Guest])
```

Configure a security-domain .

```
/subsystem=elytron/security-domain=exampleCertSD:add(realms=[{realm=ksRealm}],default-realm=ksReal
```

Configure an http-authentication-factory .

```
/subsystem=elytron/http-authentication-factory=exampleCertHttpAuth:add(http-server-mechanism-facto
```

Configure an application-security-domain in the Undertow subsystem.

```
/subsystem=undertow/application-security-domain=exampleApplicationDomain:add(http-authentication-f
```

Update server-ssl-context .

```
/subsystem=elytron/server-ssl-context=twoWaySSC:write-attribute(name=security-domain,value=example
value=true)
```

Configure your application's web.xml and jboss-web.xml .

Your application's *web.xml* and *jboss-web.xml* must be updated to use the *application-security-domain* you configured in WildFly. An example of this is available in the Configure Applications to Use Elytron or Legacy Security for Authentication section.

In addition, you need to update your *web.xml* to use *CLIENT-CERT* as its authentication method.

```
<login-config>
  <auth-method>CLIENT-CERT</auth-method>
  <realm-name>exampleApplicationDomain</realm-name>
</login-config>
```

Configure Authentication with a Kerberos-Based Identity Store

IMPORTANT: The following steps assume you have a working KDC and Kerberos domain as well as your client browsers configured.

Configure a kerberos-security-factory .

```
/subsystem=elytron/kerberos-security-factory=krbSF:add(principal="HTTP/host@REALM",path="/path/to/
```

Configure the system properties for Kerberos.

Depending on how your environment is configured, you will need to set some of the system properties below.

System Property	Description
java.security.krb5.kdc	The host name of the KDC.
java.security.krb5.realm	The name of the realm.
java.security.krb5.conf	The path to the configuration *krb5.conf* file.
sun.security.krb5.debug	If *true*, debugging mode will be enabled.

To configure a system property in WildFly:

```
/system-property=java.security.krb5.conf:add(value="/path/to/krb5.conf")
```

Configure an Eltyron security realm for assigning roles.

The the client's Kerberos token will provide the principal, but you need a way to map that principal to a role for your application. There are several ways to accomplish this, but this example creates a *filesystem-realm*, adds a user to the realm that matches the principal from the Kerberos token, and assigns roles to that user.

```
/subsystem=elytron/filesystem-realm=exampleFsRealm:add(path=fs-realm-users,relative-to=jboss.serve
value=["Admin","Guest"])
```

Add a simple-role-decoder .

```
/subsystem=elytron/simple-role-decoder=from-roles-attribute:add(attribute=Roles)
```

This *simple-role-decoder* decodes a principal's roles from the *Roles* attribute. You can change this value if your roles are in a different attribute.

Configure a security-domain .

```
/subsystem=elytron/security-domain=exampleFsSD:add(realms=[{realm=exampleFsRealm,role-decoder=from
```

Configure an http-authentication-factory that uses the kerberos-security-factory .

```
/subsystem=elytron/http-authentication-factory=example-krb-http-auth:add(http-server-mechanism-fac
```

Configure an application-security-domain in the Undertow subsystem:

```
/subsystem=undertow/application-security-domain=exampleApplicationDomain:add(http-authentication-f
```

Configure your application's web.xml , jboss-web.xml and jboss-deployment-structure.xml .

Your application's *web.xml* and *jboss-web.xml* must be updated to use the *application-security-domain* you configured in WildFly. An example of this is available in the Configure Applications to Use Elytron or Legacy Security for Authentication section.

In addition, you need to update your *web.xml* to use *SPNEGO* as its authentication method.

```
<login-config>
  <auth-method>SPNEGO</auth-method>
  <realm-name>exampleApplicationDomain</realm-name>
</login-config>
```

Configure Authentication with a Form as a Fallback for Kerberos

Configure kerberos-based authentication.

Configuring kerberos-based authentication is covered in a previous section.

Add a mechanism for FORM authentication in the http-authentication-factory .

You can use the existing *http-authentication-factory* you configured for kerberos-based authentication and and an additional mechanism for *FORM* authentication.

```
/subsystem=elytron/http-authentication-factory=example-krb-http-auth:list-add(name=mechanism-confi
value={mechanism-name=FORM})
```

Add additional fallback principals.

The existing configuration for kerberos-based authentication should already have a security realm configured for mapping principals from kerberos token to roles for the application. You can add additional users for fallback authentication to that realm. For example if you used a *filesystem-realm*, you can simply create a new user with the appropriate roles:

```
/subsystem=elytron/filesystem-realm=exampleFsRealm/identity=fallbackUser1:add()
/subsystem=elytron/filesystem-realm=exampleFsRealm/identity=fallbackUser1:set-password(clear={pass
value=["Admin","Guest"])
```

Update the web.xml for FORM fallback.

You need to update the *web.xml* to use the value *SPNEGO,FORM* for the *auth-method*, which will use *FORM* as a fallback authentication method if *SPNEGO* fails. You also need to specify the location of your login and error pages.

```
<login-config>
  <auth-method>SPNEGO,FORM</auth-method>
  <realm-name>exampleApplicationDomain</realm-name>
  <form-login-config>
    <form-login-page>/login.jsp</form-login-page>
    <form-error-page>/error.jsp</form-error-page>
  </form-login-config>
</login-config>
```

Configure Applications to Use Elytron or Legacy Security for Authentication

After you have configured the *elytron* or legacy *security* subsystems for authentication, you need to configure your application to use it.

Configure your application's web.xml .

Your application's *web.xml* needs to be configured to use the appropriate authentication method. When using *elytron*, this is defined in the *http-authentication-factory* you created. When using the legacy *security* subsystem, this depends on your login module and the type of authentication you want to configure.

Example *web.xml* with *BASIC* Authentication

```
<web-app>
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>secure</web-resource-name>
      <url-pattern>/secure/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
      <role-name>Admin</role-name>
    </auth-constraint>
  </security-constraint>
  <security-role>
    <description>The role that is required to log in to /secure/*</description>
    <role-name>Admin</role-name>
  </security-role>
  <login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>exampleApplicationDomain</realm-name>
  </login-config>
</web-app>
```

Configure your application to use a security domain.

You can configure your application's *jboss-web.xml* to specify the security domain you want to use for authentication. When using the *elytron* subsystem, this is defined when you created the *application-security-domain*. When using the legacy *security* subsystem, this is the name of the legacy security domain.

Example *jboss-web.xml*

```
<jboss-web>
  <security-domain>exampleApplicationDomain</security-domain>
</jboss-web>
```

Using *jboss-web.xml* allows you to configure the security domain for a single application only. Alternatively, you can specify a default security domain for all applications using the *undertow* subsystem. This allows you to omit using *jboss-web.xml* to configure a security domain for an individual application.

```
/subsystem=undertow:write-attribute(name=default-security-domain,
value="exampleApplicationDomain")
```

IMPORTANT: Setting *default-security-domain* in the *undertow* subsystem will apply to **ALL** applications. If *default-security-domain* is set and an application specifies a security domain in a *jboss-web.xml* file, the configuration in *jboss-web.xml* will override the *default-security-domain* in the *undertow* subsystem.

Using Elytron and Legacy Security Subsystems in Parallel

You can define authentication in both the *elytron* and legacy *security* subsystems and use them in parallel. If you use both *jboss-web.xml* and *default-security-domain* in the *undertow* subsystem, WildFly will first try to match the configured security domain in the *elytron* subsystem. If a match is not found, then WildFly will attempt to match the security domain with one configured in the legacy *security* subsystem. If the *elytron* and legacy *security* subsystem each have a security domain with the same name, the *elytron* security domain is used.

Override an Application's Authentication Configuration

You can override the authentication configuration of an application with one configured in WildFly. To do this, use the *override-deployment-configuration* property in the *application-security-domain* section of the *undertow* subsystem:

```
/subsystem=undertow/application-security-domain=exampleApplicationDomain:write-attribute(name=over
```

For example, an application is configured to use *FORM* authentication with the *exampleApplicationDomain* in its *jboss-web.xml*.

Example jboss-web.xml

```
<login-config>
  <auth-method>FORM</auth-method>
  <realm-name>exampleApplicationDomain</realm-name>
</login-config>
```

By enabling *override-deployment-configuration*, you can create a new *http-authentication-factory* that specifies a different authentication mechanism such as *BASIC*.

Example http-authentication-factory

```
/subsystem=elytron/http-authentication-factory=exampleHttpAuth:read-resource()
{
    "outcome" => "success",
    "result" => {
        "http-server-mechanism-factory" => "global",
        "mechanism-configurations" => [{
            "mechanism-name" => "BASIC",
            "mechanism-realm-configurations" => [{"realm-name" => "exampleApplicationDomain"}]
        }],
        "security-domain" => "exampleSD"
    }
}
```

This will override the authentication mechanism defined in the application's *jboss-web.xml* and attempt to authenticate a user using *BASIC* instead of *FORM*.

Create and Use a Credential Store

Create credential store.

```
/subsystem=elytron/credential-store=exampleCS:add(uri="cr-store://exampleCS?create=true",credentia
```

Add a credential to a credential store.

```
/subsystem=elytron/credential-store=exampleCS/alias=keystorepw:add(secret-value="secret")
```

List all credentials in a credential store.

```
/subsystem=elytron/credential-store=exampleCS:read-children-names(child-type=alias)
{
    "outcome" => "success",
    "result" => ["keystorepw"]
}
```

Remove a credential from a credential store.

```
/subsystem=elytron/credential-store=exampleCS/alias=keystorepw:remove
```

Use a credential store.

```
/subsystem=elytron/key-store=twoWayKS:write-attribute(name=credential-reference,value={store=examp
```

15.14.2 Set up and Configure Authentication for the Management Interfaces

Secure the Management Interfaces with a New Identity Store

Create a security domain and any supporting security realms, decoders, or mappers for your identity store.

This process is covered in a previous section. For example, if you wanted to secure the management interfaces using a filesystem-based identity store, you would follow the steps in Configure Authentication with a Filesystem-Based Identity Store.

Create an http-authentication-factory or sasl-authentication-factory .

Example *http-authentication-factory*

```
/subsystem=elytron/http-authentication-factory=example-http-auth:add(http-server-mechanism-factory
```

Example *sasl-authentication-factory*

```
/subsystem=elytron/sasl-authentication-factory=example-sasl-auth:add(sasl-server-factory=configure
```

Update the management interfaces to use your http-authentication-factory or sasl-authentication-factory .

Example update *http-authentication-factory*

```
/core-service=management/management-interface=http-interface:write-attribute(name=http-authenticat
value=example-http-auth)
{
   "outcome" => "success",
   "response-headers" => {
       "operation-requires-reload" => true,
       "process-state" => "reload-required"
   }
}

reload
```

Example update *sasl-authentication-factory*

```
/core-service=management/management-interface=http-interface:write-attribute(name=http-upgrade.sas
value=example-sasl-auth)
{
   "outcome" => "success",
   "response-headers" => {
       "operation-requires-reload" => true,
       "process-state" => "reload-required"
   }
}

reload
```

Silent Authentication

By default, WildFly provides an authentication mechanism for local users, also know as silent authentication, through the *local* security realm.

Silent authentication must be used via a *sasl-authentication-factory*.

IMPORTANT: When enabling silent authentication, you must ensure the security domain referenced by your *sasl-authentication-factory* references a security realm that contains the *$local* user. By default, WildFly provides the *local* identity realm that provides this user.

Add silent authentication to an existing sasl-authentication-factory .

```
/subsystem=elytron/sasl-authentication-factory=example-sasl-auth:list-add(name=mechanism-configura
value={mechanism-name=JBOSS-LOCAL-USER, realm-mapper=local})

reload
```

Create a new sasl-server-factory with silent authentication.

```
/subsystem=elytron/sasl-authentication-factory=example-sasl-auth:add(sasl-server-factory=configure
realm-mapper=local}])

reload
```

Remove silent authentication from an existing sasl-server-factory :

```
/subsystem=elytron/sasl-authentication-factory=managenet-sasl-authentication:read-resource
{
    "outcome" => "success",
    "result" => {
        "mechanism-configurations" => [
            {
                "mechanism-name" => "JBOSS-LOCAL-USER",
                "realm-mapper" => "local"
            },
            {
                "mechanism-name" => "DIGEST-MD5",
                "mechanism-realm-configurations" => [{"realm-name" => "ManagementRealm"}]
            }
        ],
        "sasl-server-factory" => "configured",
        "security-domain" => "ManagementDomain"
    }
}

/subsystem=elytron/sasl-authentication-factory=temp-sasl-authentication:list-remove(name=mechanism
```

Using RBAC with Elytron

RBAC can be configured to automatically assign or exclude roles for users that are members of groups. This is configured in the *access-control* section of the core management. When the management interfaces are secured with the *elytron* subsystem, and users are assigned groups when they authenticate. You can also configure roles to be assigned to authenticated users in a variety of ways using the *elytron* subsystem, for example using a role mapper or a role decoder.

15.14.3 Configure SSL/TLS

Enable One-way SSL/TLS for Applications

In WildFly, you can use the Elytron subsystem, along with the Undertow subsystem, to enable HTTPS for deployed applications.

Obtain or generate your key store:

Before enabling HTTPS in WildFly, you must obtain or generate the keystore you plan on using. To generate an example keystore:

```
$ keytool -genkeypair -alias localhost -keyalg RSA -keysize 1024 -validity 365 -keystore
/path/to/keystore.jks -dname "CN=localhost" -keypass secret -storepass secret
```

Configure a key-store in WildFly:

```
/subsystem=elytron/key-store=httpsKS:add(path=/path/to/keystore.jks,credential-reference={clear-te
```

The previous command uses an absolute path to the keystore. Alternatively you can use the *relative-to* attribute to specify the base directory variable and *path* specify a relative path.

```
/subsystem=elytron/key-store=httpsKS:add(path=keystore.jks,relative-to=jboss.server.config.dir,cre
```

Configure a key-manager in that references your key-store :

```
/subsystem=elytron/key-manager=httpsKM:add(key-store=httpsKS,credential-reference={clear-text=secr
```

Configure a server-ssl-context in that references your key-manager :

```
/subsystem=elytron/server-ssl-context=httpsSSC:add(key-manager=httpsKM,protocols=["TLSv1.2"])
```

IMPORTANT: You need to determine what SSL/TLS protocols you want to support. The example commands above uses *TLSv1.2*.

Check and see if the https-listener is configured to use a legacy security realm for its SSL configuration:

```
/subsystem=undertow/server=default-server/https-listener=https:read-attribute(name=security-realm)
"outcome" => "success",
    "result" => "ApplicationRealm"
}
```

The above command shows that the *https-listener* is configured to use the *ApplicationRealm* legacy security realm for its SSL configuration. Undertow cannot reference both a legacy security realm and an *ssl-context* in Elytron at the same time so you must remove the reference to the legacy security realm. Also there has to be always configured either *ssl-context* or *security-realm*. Thus when changing between those, you have to use batch operation:

Remove the reference to the legacy security realm and update the *https-listener* to use the *ssl-context* from Elytron :

```
batch
/subsystem=undertow/server=default-server/https-listener=https:undefine-attribute(name=security-re
```

Reload the server:

```
reload
```

HTTPS is now enabled for applications.

Enable Two-way SSL/TLS in WildFly for Applications

In WildFly, you can use the Elytron subsystem, along with the Undertow subsystem, to enable two-way SSL/TLS for deployed applications.

Obtain or generate your keystore:

Before enabling HTTPS in WildFly, you must obtain or generate the keystores, truststores and certificates you plan on using.

Create server and client keystores:

```
$ keytool -genkeypair -alias localhost -keyalg RSA -keysize 1024 -validity 365 -keystore
server.keystore.jks -dname "CN=localhost" -keypass secret -storepass secret

$ keytool -genkeypair -alias client -keyalg RSA -keysize 1024 -validity 365 -keystore
client.keystore.jks -dname "CN=client" -keypass secret -storepass secret
```

Export the server and client certificates:

```
$ keytool -exportcert  -keystore server.keystore.jks -alias localhost -keypass secret -storepass
secret -file server.cer

$ keytool -exportcert  -keystore client.keystore.jks -alias client -keypass secret -storepass
secret -file client.cer
```

Import the sever and client certificates into the opposing truststores:

```
$ keytool -importcert -keystore server.truststore.jks -storepass secret -alias client
-trustcacerts -file client.cer

$ keytool -importcert -keystore client.truststore.jks -storepass secret -alias localhost
-trustcacerts -file server.cer
```

Configure a key-store for server keystore and truststore in WildFly:

```
/subsystem=elytron/key-store=twoWayKS:add(path=/path/to/server.keystore.jks,credential-reference={
```

NOTE

The previous command uses an absolute path to the keystore. Alternatively you can use the *relative-to* attribute to specify the base directory variable and *path* specify a relative path.

```
/subsystem=elytron/key-store=myKS:add(path=keystore.jks,relative-to=jboss.server.config.dir,creden
```

Configure a key-manager in that references your key store key-store :

```
/subsystem=elytron/key-manager=twoWayKM:add(key-store=twoWayKS,credential-reference={clear-text=se
```

Configure a trust-manager in that references your truststore key-store :

```
/subsystem=elytron/trust-manager=twoWayTM:add(key-store=twoWayTS)
```

Configure a server-ssl-context in that references your key-manager , trust-manager , and enables client authentication:

```
/subsystem=elytron/server-ssl-context=twoWaySSC:add(key-manager=twoWayKM,protocols=["TLSv1.2"],tru
```

IMPORTANT
You need to determine what SSL/TLS protocols you want to support. The example commands above uses *TLSv1.2*.

Check and see if the https-listener is configured to use a legacy security realm for its SSL configuration:

```
/subsystem=undertow/server=default-server/https-listener=https:read-attribute(name=security-realm)
"outcome" => "success",
    "result" => "ApplicationRealm"
}
```

The above command shows that the *https-listener* is configured to use the *ApplicationRealm* legacy security realm for its SSL configuration. Undertow cannot reference both a legacy security realm and an *ssl-context* in Elytron at the same time so you must remove the reference to the legacy security realm. Also there has to be always configured either *ssl-context* or *security-realm*. Thus when changing between those, you have to use batch operation:

Remove the reference to the legacy security realm and update the https-listener to use the ssl-context from Elytron:

```
batch
/subsystem=undertow/server=default-server/https-listener=https:undefine-attribute(name=security-re
```

Reload the server

```
reload
```

Configure your client to use the client certificate

You need to configure your client to present the trusted client certificate to the server to complete the two-way SSL/TLS authentication. For example, if using a browser, you need to import the trusted certificate into the browser's truststore.

Two-Way HTTPS is now enabled for applications.

Enable One-way SSL/TLS for the Management Interfaces Using the Elytron Subsystem

Obtain or generate your key store:

Before enabling HTTPS in WildFly, you must obtain or generate the key store you plan on using. To generate an example key store, use the following command.

```
$ keytool -genkeypair -alias localhost -keyalg RSA -keysize 1024 -validity 365 -keystore
keystore.jks -dname "CN=localhost" -keypass secret -storepass secret
```

Create a key-store , key-manager , and server-ssl-context .

```
/subsystem=elytron/key-store=httpsKS:add(path=keystore.jks,relative-to=jboss.server.config.dir,cre
```

IMPORTANT: You need to determine what SSL/TLS protocols you want to support. The example commands above uses *TLSv1.2*.

NOTE: The above command uses *relative-to* to reference the location of the keystore file. Alternatively, you can specify the full path to the keystore in *path* and omit *relative-to*.

Enable HTTPS on the management interface.

```
/core-service=management/management-interface=http-interface:write-attribute(name=ssl-context,
value=httpsSSC)

/core-service=management/management-interface=http-interface:write-attribute(name=secure-socket-bi
value=management-https)
```

Reload the WildFly instance.

```
reload
```

HTTPS is now enabled for the management interfaces.

Enable Two-way SSL/TLS for the Management Interfaces using the Elytron Subsystem

Obtain or generate your key store.

Before enabling HTTPS in WildFly, you must obtain or generate the key stores, trust stores and certificates you plan on using. To generate an example set of key stores, trust stores, and certificates use the following commands.

Generate your server and client key stores.

```
$ keytool -genkeypair -alias localhost -keyalg RSA -keysize 1024 -validity 365 -keystore
server.keystore.jks -dname "CN=localhost" -keypass secret -storepass secret

$ keytool -genkeypair -alias client -keyalg RSA -keysize 1024 -validity 365 -keystore
client.keystore.jks -dname "CN=client" -keypass secret -storepass secret
```

Export your server and client certificates.

```
$ keytool -exportcert  -keystore server.keystore.jks -alias localhost -keypass secret -storepass
secret -file server.cer

$ keytool -exportcert  -keystore client.keystore.jks -alias client -keypass secret -storepass
secret -file client.cer
```

Import the sever and client certificates into the opposing trust stores.

```
$ keytool -importcert -keystore server.truststore.jks -storepass secret -alias client
-trustcacerts -file client.cer

$ keytool -importcert -keystore client.truststore.jks -storepass secret -alias localhost
-trustcacerts -file server.cer
```

Configure key-store , a key-manager , trust-manager , and server-ssl-context for the server key store and trust store.

```
/subsystem=elytron/key-store=twoWayKS:add(path=server.keystore.jks,relative-to=jboss.server.config
```

IMPORTANT: You need to determine what SSL/TLS protocols you want to support. The example commands above uses *TLSv1.2*.

NOTE: The above command uses *relative-to* to reference the location of the keystore file. Alternatively, you can specify the full path to the keystore in *path* and omit *relative-to*.

Enable HTTPS on the management interface.

```
/core-service=management/management-interface=http-interface:write-attribute(name=ssl-context,
value=twoWaySSC)

/core-service=management/management-interface=http-interface:write-attribute(name=secure-socket-bi
value=management-https)
```

Reload the WildFly instance.

```
reload
```

Configure your client to use the client certificate.

You need to configure your client to present the trusted client certificate to the server to complete the two-way SSL/TLS authentication. For example, if using a browser, you need to import the trusted certificate into the browser's trust store.

Two-way SSL/TLS is now enabled for the management interfaces.

Using an ldap-key-store

An *ldap-key-store* allows you to use a keystore stored in an LDAP server. You can use an *ldap-key-store* in same way you can use a *key-store*.

To create and use an *ldap-key-store*:

Configure a dir-context .

To connect to the LDAP server from WildFly, you need to configure a *dir-context* that provides the URL as well as the principal used to connect to the server.

Example dir-context

```
/subsystem=elytron/dir-context=exampleDC:add( \
  url="ldap://127.0.0.1:10389", \
  principal="uid=admin,ou=system", \
  credential-reference={clear-text=secret} \
)
```

Configure an ldap-key-store .

When configure an *ldap-key-store*, you need to specify both the *dir-context* used to connect to the LDAP server as well as how to locate the keystore stored in the LDAP server. At a minimum, this requires you specify a *search-path*.

Example ldap-key-store

```
/subsystem=elytron/ldap-key-store=ldapKS:add( \
  dir-context=exampleDC, \
  search-path="ou=Keystores,dc=wildfly,dc=org" \
)
```

Use the ldap-key-store .

Once you have defined your *ldap-key-store*, you can use it in the same places where a *key-store* could be used. For example, you could use an *ldap-key-store* when configuring HTTPS and Two-Way HTTPS for applications.

Using a filtering-key-store

A *filtering-key-store* allows you to expose a subset of aliases from an existing *key-store*, and use it in the same places you could use a *key-store*. For example, if a keystore contained *alias1*, *alias2*, and *alias3*, but you only wanted to expose *alias1* and *alias3*, a *filtering-key-store* provides you several ways to do that.

To create a *filtering-key-store*:

Configure a key-store .

```
/subsystem=elytron/key-store=myKS:add( \
  path=keystore.jks, \
  relative-to=jboss.server.config.dir, \
  credential-reference={ \
    clear-text=secret \
  }, \
  type=JKS \
)
```

Configure a filtering-key-store .

When you configure a *filtering-key-store*, you specify which *key-store* you want to filter and the *alias-filter* for filtering aliases from the *key-store*. The filter can be specified in one of the following formats:

- *alias1,alias3*, which is a comma-delimited list of aliases to expose.
- *ALL:-alias2*, which exposes all aliases in the keystore except the ones listed.
- *NONE:+alias1:+alias3*, which exposes no aliases in the keystore except the ones listed.

This example uses a comma-delimted list to expose *alias1* and *alias3*.

```
/subsystem=elytron/filtering-key-store=filterKS:add( \
  key-store=myKS, \
  alias-filter="alias1,alias3" \
)
```

Use the filtering-key-store .

Once you have defined your *filtering-key-store*, you can use it in the same places where a *key-store* could be used. For example, you could use a *filtering-key-store* when configuring HTTPS and Two-Way HTTPS for applications.

Reload a Keystore

You can reload a keystore configured in WildFly from the management CLI. This is useful in cases where you have made changes to certificates referenced by a keystore.

To reload a keystore.

```
/subsystem=elytron/key-store=httpsKS:load
```

Check the Content of a Keystore by Alias

If you add a keystore to the *elytron* subsystem using the *key-store* component, you can check the keystore's contents using the *alias* child element and reading its attributes.

For example:

```
/subsystem=elytron/key-store=httpsKS/alias=localhost:read-attribute(name=certificate-chain)
{
    "outcome" => "success",
    "result" => [{
        "type" => "X.509",
        "algorithm" => "RSA",
        "format" => "X.509",
        "public-key" => "30:81:9f:30:0d:06:09:2a:8.....
```

The following attributes can be read:

Attribute	Description
certificate	The certificate associated with the alias. If the alias has a certificate chain this will always be undefined.
certificate-chain	The certificate chain associated with the alias.
creation-date	The creation date of the entry represented by this alias.
entry-type	The type of the entry for this alias. Available types: *PasswordEntry*, *PrivateKeyEntry*, *SecretKeyEntry*, *TrustedCertificateEntry*, and *Other*. Unrecognized types will be reported as *Other*.

Custom Components

When configuring SSL/TLS in the *elytron* subsystem, you can provide and use custom implementations of the following components:

- *key-store*
- *key-manager*
- *trust-manager*
- *client-ssl-context*
- *server-ssl-context*

When creating custom implementations of Elytron components, they must present the appropriate capabilities and requirements.

15.14.4 Configuring the Elytron and Security Subsystems

Enable and Disable the Elytron Subsystem

To add the elytron extension required for the elytron subsystem:

```
/extension=org.wildfly.extension.elytron:add()
```

To enable the Elytron subsystem in WildFly:

```
/subsystem=elytron:add

reload
```

To disable the Elytron subsystem in WildFly:

```
/subsystem=elytron:remove

reload
```

IMPORTANT: Other subsystems within WildFly may have dependencies on the *elytron* subsystem. If these dependencies are not resolved before disabling it, you will see errors when starting WildFly.

Enable and Disable the Security Subsystem

To disable the security subsystem in WildFly:

```
/subsystem=security:remove

reload
```

IMPORTANT: Other subsystems within WildFly may have dependencies on the *security* subsystem. If these dependencies are not resolved before disabling it, you will see errors when starting WildFly.

To enable the security subsystem in WildFly:

```
/subsystem=security:add

reload
```

Use the Elytron and Security Subsystems in Parallel

By default the *elytron* and *security* subsystems will run in parallel if both are enabled. For authentication in applications, you can use the *application-security-domain* property in the *undertow* subsystem to configure a security domain in the *elytron* subsystem.

```
/subsystem=undertow/application-security-domain=exampleApplicationDomain:add(http-authentication-f
```

NOTE: This must match the *security-domain* configured in the *jboss-web.xml* of your application.

If the *application-security-domain* is not set, WildFly will look for a security domain configured in the *security* subsystem that matches the *security-domain* configured in the *jboss-web.xml* of your application.

For enabling HTTPS using a legacy security realm, you can use the *security-realm* attribute in the *https-listener* section of the *undertow* subsystem:

```
/subsystem=undertow/server=default-server/https-listener=https:read-attribute(name=security-realm)
"outcome" => "success",
    "result" => "ApplicationRealm"
}
```

For enabling HTTPS using *elytron*, you need to undefine the *security-realm* attribute and set the *ssl-context* attribute. As there has to be always configured either *ssl-context* or *security-realm* you have to use batch operation when changing between those:

```
batch
/subsystem=undertow/server=default-server/https-listener=https:undefine-attribute(name=security-re
```

15.14.5 Creating Elytron Subsystem Components

Create an Elytron Security Realm

Security realms in the Elytron subsystem, when used in conjunction with security domains, are use for both core management authentication as well as for authentication with applications. Security realms are also specifically typed based on their identity store, for example *jdbc-realm*, *filesystem-realm*, *properties-realm*, etc.

Adding a security realm takes the general form:

```
/subsystem=elytron/type-of-realm=realmName:add(....)
```

Examples of adding specific realms, such as *jdbc-realm*, *filesystem-realm*, and *properties-realm* can be found in previous sections.

Create an Elytron Role Decoder

A role decoder converts attributes from the identity provided by the security realm into roles. Role decoders are also specifically typed based on their functionality, for example *empty-role-decoder*, *simple-role-decoder*, and *custom-role-decoder*.

Adding a role decoder takes the general form:

```
/subsystem=elytron/ROLE-DECODER-TYPE=roleDeoderName:add(....)
```

Create an Elytron Permission Mapper

In addition to roles being assigned to a identity, permissions may also be assigned. A permission mapper assigns permissions to an identity. Permission mappers are also specifically typed based on their functionality, for example *logical-permission-mapper*, *simple-permission-mapper*, and *custom-permission-mapper*.

Adding a permission mapper takes the general form:

```
/subsystem=elytron/simple-permission-mapper=PermissionMapperName:add(...)
```

Create an Elytron Role Mapper

A role mapper maps roles after they have been decoded to other roles. Examples include normalizing role names or adding and removing specific roles from principals after they have been decoded. Role mappers are also specifically typed based on their functionality, for example *add-prefix-role-mapper*, *add-suffix-role-mapper*, and *constant-role-mapper*.

Adding a role mapper takes the general form:

```
/subsystem=elytron/ROLEM-MAPPER-TYPE=roleMapperName:add(...)
```

Create an Elytron Security Domain

Security domains in the Elytron subsystem, when used in conjunction with security realms, are use for both core management authentication as well as for authentication with applications.

Adding a security domain takes the general form:

```
/subsystem=elytron/security-domain=domainName:add(realms=[{realm=realmName,role-decoder=roleDecode
```

Create an Elytron Authentication Factory

An authentication factory is an authentication policy used for specific authentication mechanisms. Authenticaion factories are specifically based on the authentication mechanism, for example *http-authentication-factory* and *sasl-authentication-factory* and *kerberos-security-factory*.

Adding an authentication factory takes the general form:

```
/subsystem=elytron/AUTH-FACTORY-TYPE=authFactoryName:add(....)
```

Create an Elytron Policy Provider

Elytron subsystem provides a specific resource definition that can be used to configure a default Java Policy provider. The subsystem allows you to define multiple policy providers but select a single one as the default:

```
/subsystem=elytron/policy=policy-provider-a:add(custom-policy=\[{name=policy-provider-a,
class-name=MyPolicyProviderA, module=x.y.z}\])
```

```

```

15.15 Using Elytron within WildFly

15.15.1 Using the Out of the Box Elytron Components

Securing Management Interfaces

You can find more details on the enabling WildFly to use the out of the box Elytron components for securing the management interfaces in the Default Management Authentication Configuration section.

Securing Applications

The *elytron* subsystem provides *application-http-authentication* by default which can be used to secure applications. For more details on how *application-http-authentication* is configured, see the Out of the Box Configuration section.

To configure applications to use *application-http-authentication*, see Configure Applications to Use Elytron or Legacy Security for Authentication. You can also override the default behavior of all applications using the steps in Override an Application's Authentication Configuration.

Using SSL/TLS

WildFly does provide a default one-way SSL/TLS configuration using the legacy core management authentication but does not provide one in the *elytron* subsystem. You can find more details on configuring SSL/TLS using the *elytron* subsystem for both the management interfaces as well as for applications in Configure SSL/TLS

Using Elytron with Other Subsystems

In addition to securing applications and management interfaces, Elytron also integrates with other subsystems in WildFly.

Subsystem	Details
batch-jberet	You can configure the *batch-jberet* to run batch jobs using an Elytron security domain.
datasources	You can use a credential store or an Elytron security domain to provide authentication information in a datasource definition.
messaging-activemq	You can secure remote connections to the remote connections used by the *messaging-activemq* subsystem.
iiop-openjdk	You can use the *elytron* subsystem to configure SSL/TLS between clients and servers using the *iiop-openjdk* subsystem.
mail	You can use a credential store to provide authentication information in a server definition in the *mail* subsystem.
undertow	You can use the *elytron* subsystem to configure both SSL/TLS and application authentication.

15.15.2 Undertow Subsystem

15.15.3 EJB Subsystem

Configuration can be added to the EJB subsystem to map a security domain name referenced in a deployment to an Elytron security domain:

```
/subsystem=ejb3/application-security-domain=MyAppSecurity:add(security-domain=ApplicationDomain)
```

Which results in:

```
<subsystem xmlns="urn:jboss:domain:ejb3:5.0">
...
    <application-security-domains>
        <application-security-domain name="MyAppSecurity" security-domain="ApplicationDomain"/>
    </application-security-domains>
...
</subsystem>
```

Note: If the deployment was already deployed at this point the application server should be reloaded or the deployment redeployed for the application security domain mapping to take effect.

An `application-security-domain` has two main attributes:

- name - the name of the security domain as specified in a deployment
- security-domain - a reference to the Elytron security domain that should be used

When an application security domain mapping is configured for a bean in a deployment, this indicates that security should be handled by Elytron.

15.15.4 WebServices Subsystem

There is adapter in webservices subsystem to make authentication works for elytron security domain automatically. Like configure with legacy security domain, you can configure elytron security domain in deployment descriptor or annotation to secure webservice endpoint.

15.15.5 Legacy Security Subsystem

As previously described, Elytron based security is configured by chaining together different capability references to form a complete security policy. To allow an incremental migration from the legacy Security subsystem some of the major components of this subsystem can be mapped to Elytron capabilities and used within an Elytron based set up.

15.16 Web Single Sign-On

15.16.1 Overview

This document will guide on how to enable single sign-on across different applications deployed into different servers, where these applications belong to same security domain.

15.16.2 Create a Server Configuration Template

For this document, you'll need to run at least two server instances in order to check single sign-on and how it affect usability in your applications. Users should be able to log in once and have access to any application using the same security domain.

All configuration described in the next sections should be done with a server instance using **standalone-ha.xml** (or standalone-full-ha.xml).

Run a server instance using the following command:

```
bin/standalone.sh -c standalone-ha.xml
```

Create a HTTP Authentication Factory

If you already have a *http-authentication-factory* defined in Elytron subsystem and just want to use it to enable single sign-on to your applications, please skip this section.

First, you need a *security-domain* which we'll use to authenticate users. Please, execute the following CLI commands:

```
# Creates a FileSystem Realm, an identity store where users are stored in the local filesystem
/subsystem=elytron/filesystem-realm=example-realm:add(path=/tmp/example-realm)

# Creates a Security Domain
/subsystem=elytron/security-domain=example-domain:add(default-realm=example-realm,
permission-mapper=default-permission-mapper,realms=[{realm=example-realm,
role-decoder=groups-to-roles}]

# Creates an user that you can use to access your applications
/subsystem=elytron/filesystem-realm=example-realm:add-identity(identity=alice)
/subsystem=elytron/filesystem-realm=example-realm:add-identity-attribute(identity=alice,
name=groups, value=["user"])
/subsystem=elytron/filesystem-realm=example-realm:set-password(identity=alice,
clear={password=alice})
```

Now you can create a *http-authentication-factory* that you'll use to actually protect your web applications using Undertow:

```
# Create a Http Authentication Factory
/subsystem=elytron/http-authentication-factory=example-http-authentication:add(security-domain=exa
http-server-mechanism-factory=global, mechanism-configurations=[{mechanism-name=FORM}]
```

Create a Application Security Domain in Undertow

If you already have a *application-security-domain* defined in Undertow subsystem and just want to use it to enable single sign-on to your applications, please skip this section.

In order to protect applications using the configuration defined in Elytron subsystem, you should create a application-security-domain definition in Undertow subsystem as follows:

```
/subsystem=undertow/application-security-domain=other:add(http-authentication-factory=example-http
```

By default, if your application does not define any specific *security-domain* in *jboss-web.xml*, the application server will choose one with a name **other**.

Create a Key Store

In order to create a *key-store* in Elytron subsystem, first create a Java Key Store as follows:

```
keytool -genkeypair -alias localhost -keyalg RSA -keysize 1024 -validity 365 -keystore
keystore.jks -dname "CN=localhost" -keypass secret -storepass secret
```

Once the *keystore.jks* file is created, execute the following CLI commands to create a *key-store* definition in Elytron:

```
/subsystem=elytron/key-store=example-keystore:add(path=keystore.jks,
relative-to=jboss.server.config.dir, credential-reference={clear-text=secret}, type=JKS)
```

Enable Single Sign-On

Single Sign-On is enabled to a specific *application-security-domain* definition in Undertow subsystem. It is important that the servers you will be using to deploy applications are using the same configuration.

To enable single-sign on, just change an existing application-security-domain in Undertow subsystem as follows:

```
/subsystem=undertow/application-security-domain=other/setting=single-sign-on:add(key-store=example
key-alias=localhost, domain=localhost, credential-reference={clear-text=secret})
```

After restarting the servers, users should be able to log in once and have access to any application using the same *application-security-domain*.

15.16.3 Create Two Server Instances

All configuration you did so far should be reflect in *$JBOSS_HOME/standalone/standalone-ha.xml.* You can now create two distinct server configuration directories_:_

```
cp -r standalone standalone-a
cp -r standalone standalone-b
```

And you can run the two instances using the command below:

```
$JBOSS_HOME/bin/standalone.sh -c standalone-ha.xml -Djboss.node.name=node-a
-Djboss.socket.binding.port-offset=200 -Djboss.server.base.dir=$JBOSS_HOME/standalone-a
$JBOSS_HOME/bin/standalone.sh -c standalone-ha.xml -Djboss.node.name=node-b
-Djboss.socket.binding.port-offset=300 -Djboss.server.base.dir=$JBOSS_HOME/standalone-b
```

15.16.4 Deploy an Application

For the sake of simplicity, these are the minimum files you need in your application:

WEB-INF/web.xml

```
<web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd">

    <security-constraint>
        <display-name>SecurityConstraint</display-name>
        <web-resource-collection>
            <web-resource-name>All Resources</web-resource-name>
            <url-pattern>/*</url-pattern>
        </web-resource-collection>
        <auth-constraint>
            <role-name>user</role-name>
        </auth-constraint>
    </security-constraint>

    <login-config>
        <auth-method>FORM</auth-method>
        <form-login-config>
            <form-login-page>/login.html</form-login-page>
            <form-error-page>/login.html</form-error-page>
        </form-login-config>
    </login-config>

    <security-role>
        <role-name>user</role-name>
    </security-role>
</web-app>
```

login.html

```
<html>
    <body>
        <form method="post" action="j_security_check">
            <input type="text" name="j_username">
            <input type="password" name="j_password">
            <input type="submit" value="Log In">
        </form>
    </body>
</html>
```

Make sure you have at least a welcome file (e.g.: index.html|jsp).

Deploy your application into both server instances and try to log in using the user you created at the beginning of this document:

- Username: **alice**
- Password: **alice**

16 WildFly Client Configuration

16.1 Introduction

As of WildFly 11 a common configuration framework has been introduced for use by the client libraries to define configuration, this allows for the configuration to be shared across multiple clients rather than relying on their own configuration files. As an example the configuration used by an EJB client can be shared with the JBoss CLI, if both of these required SSL configuration this can now be defined once and re-used.

Programatic APIs are also available for many of the options however this document is focusing on the configuration available within the common *wildfly-config.xml* configuration file.

16.1.1 wildfly-config.xml Discovery

At the time a client requires access to its configuration, the class path is scanned for a wildfly-config.xml or META-INF/wildfly-config.xml file. Once the file is located the configuration will be parsed to be made available for that client.

Alternatively, the *wildfly.config.url* system property can also be specified to identify the location of the configuration that should be used.

16.2 Configuration Sections

16.2.1 <authentication-client /> - WildFly Elytron

The <*authentication-client/*> element can be added to the wildfly-config.xml configuration to define configuration in relation to authentication configuration for outbound connections and SSL configuration for outbound connections e.g.

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration>
    <authentication-client xmlns="urn:elytron:1.0">
       ...
    </authentication-client>
</configuration>
```

Note: A single wildfly-config.xml could be used by multiple projects using multiple versions of Wildfly Elytron, newer versions of WildFly Elytron will introduce new configuration using later namespace versions however they will still continue to support the previously released schemas until advertised otherwise. For the configuration to be compatible with this either use a schema and namespace compatible with all the versions in use, or multiple authentication-client elements can be added, these should be added ordered by namespace youngest to oldest. If a configuration with a later namespace is discovered by a newer WildFly Elytron client it will be used and parsing will not look for an older version as well.

The <*authentication-client* /> configuration can contain the following sections: -

- **<credential-stores />** - Definitions of credential stores to be referenced from elsewhere in the configuration.
- **<key-stores />** - Definitions of KeyStores to be referenced elsewhere in the configuration.
- **<authentication-rules />** - Rules to be applied for outbound connections to match against an appropriate authentication configuration.
- **<authentication-configurations />** - The individual authentication configurations that will be matched by the authentication rules.
- **<net-authenticator />** - Flag to enable integration with the java.net.Authenticator.
- **<ssl-context-rules />** - Rules to be applied for outbound connections to match against an appropriate SSL context configuration.
- **<ssl-contexts />** - Individual SSL context definitions that will be matched by the ssl context rules.
- **<providers/>** - Definition of how java.security.Provider instances will be discovered.

<credential-stores />

The <credential-stores /> element can be used to define individual named credential stores that can subsequently be used elsewhere in the configuration to reference stored credentials as an alternative to the credentials being embedded in the configuration.

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration>
    <authentication-client xmlns="urn:elytron:1.0">
        <credential-stores>
            <credential-store name="..." type="..." provider="..." >
                <attributes>
                    <attribute name="..." value="..." />
                </attributes>
                <protection-parameter-credentials>...</protection-parameter-credentials>
            </credential-store>
        </credential-stores>
    </authentication-client>
</configuration>
```

In addition to the name an individual <credential-store /> definition can contain the following optional attributes: -

1. **type** - The type of the credential store, e.g.KeyStoreCredentialStore.
2. **provider** - The name of the java.security.Provider to use to load the credential store.

The following child elements can also be added to configure the credential store.

- **<attributes .>** - Definition of configuration attributes used to initialise the credential store.

e.g.

```
<attributes>
    <attribute name="..." value="..." />
</attributes>
```

The <attribute/> element can be repeated as many times as is required for the configuration.

- **<protection-parameter-credentials />** - One or more credentials to be assembled into a protection parameter when initialising the credential store.

The <protection-paramter-credentials /> element can contain one more more child elements, which of these are actually supported will depend on the credential store implementation: -

```
<protection-parameter-credentials>
    <key-store-reference>...</key-store-reference>
    <credential-store-reference store="..." alias="..." clear-text="..." />
    <clear-password password="..." />
    <key-pair public-key-pem="..." private-key-pem="..." />
    <certificate private-key-pem="..." pem="..." />
    <public-key-pem>...</public-key-pem>
    <bearer-token value="..." />
    <oauth2-bearer-token token-endpoint-uri="...">...</oauth2-bearer-token>
</protection-parameter-credentials>
```

The potential child elements of <protection-parameter-credentials /> are: -

- **<key-store-reference>** - Defines a reference to an entry within a KeyStore for an entry to use.

The overall structure of this element is: -

```
<key-store-reference key-store-name="..." alias="...">
   <key-store-clear-password password="..." />
   <credential-store-reference store="..." alias="..." clear-text="..." />
   <key-store-credential>...</key-store-credential>
</key-store-reference>
```

This structure is identical to the structure use in <key-store-credential />.

- **<credential-store-reference store="..." alias="..." clear-text="..." />** - Reference to a credential stored in a credential store.
- **<clear-password password="..." />** - A password specified in the clear.
- **<key-pair public-key-pem="..." private-key-pem="..." />** - A public and private key pair.
- <certificate private-key-pem="..." pem="..." />* - A pem encoded private key and corresponding certificate.
- **<public-key-pem>...</public-key-pem>** - A pem encoded public key.

- <bearer-token value="..." />* - A bearer token
- **<oauth2-bearer-token>...</oauth2-bearer-token>** - An oath2 bearer token.

The full structure of this element is: -

```
<oauth2-bearer-token token-endpoint-uri="...">
    <client-credentials client-id="..." client-secret="..." />
    <resource-owner-credentials name="..." password="..." />
</oauth2-bearer-token>
```

<key-stores />

The <key-stores /> element can be used to define individual key-store definitions that can subsequently be referenced from alternative locations within the configuration.

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration>
    <authentication-client xmlns="urn:elytron:1.0">
        <key-stores>
            <key-store name="...">
               <!-- One of the following to specify where to load the KeyStore from. -->
               <file-name name="..." />
               <load-from uri-"..." />
               <resource name="..." />
               <!-- One of the following to specify the protection parameter to unlock the
KeyStore. -->
               <key-store-clear-password password="..." />
               <key-store-credential>...</key-store-credential>
            </key-store>
        </key-stores>
       ...
    </authentication-client>
</configuration>
```

An individual <key-store /> definition must contain exactly one of the following elements to define where to load the store from.

1. **<file name="..." />** - Load from file where 'name' is the name of the file.
2. **<load-from uri="..." />** - Load the file from the URI specified.
3. **<resource name="..." />** - Load as a resource from the Thread context classloader where 'name' is the name of the resource to load.

Exactly one of the following elements must also be present to specify the protection parameter for initialisation of the KeyStore.

1. **<key-store-clear-password password="..." />** - A password specified in the clear.
2. **<key-store-credential>...</key-store-credential>** - A reference to another KeyStore to obtain an Entry to use as the protection parameter to access this KeyStore.

The structure of the <key-store-credential /> element is.

```
<key-store-credential key-store-name="..." alias="...">
   <key-store-clear-password password="..." />
   <credential-store-reference store="..." alias="..." clear-text="..." />
   <key-store-credential>...</key-store-credential>
</key-store-credential>
```

This element contains two attributes: -

1. **key-store-name** *(Mandatory)* - Name of the KeyStore being referenced to load the entry from.
2. **alias** *(Optional)* - The alias of the entry to load from the referenced KeyStore, this can only be omitted for KeyStores that contain only a single entry.

Java KeyStores also make use of a protection parameter when accessing a single entry in addition to the protection parameter to load the KeyStore, exactly one of the following elements must be present to specify the protection parameter of the entry being loaded.

1. **<key-store-clear-password password="..." />** - A password specified in the clear.
2. **<credential-store-reference store="..." alias="..." clear-text="..." />** - Reference to a credential stored in a credential store.
3. **<key-store-credential>...</key-store-credential>** - A reference to another KeyStore to obtain an Entry to use as the protection parameter to access the alias.

The <key-store-credential /> is exactly the same, this means theoretically a chain of references could be used to lead to the unlocking of the required alias.

<authentication-rules /> and <ssl-context-rules />

When either an authentication-configuration or an ssl-context is required the URI of the resources being accessed as well as an optional abstract type and abstract type authority and matched against the rules defined in the configuration to identify which authentication-configuration or ssl-context should be used.

The rules to match <authentication-configuration /> instances are defined within the <authentication-rules /> element.

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration>
    <authentication-client xmlns="urn:elytron:1.0">
        <authentication-rules>
            <rule use-configuration="...">
                ...
            </rule>
        </authentication-rules>
       ...
    </authentication-client>
</configuration>
```

The rules to match against the <ssl-context /> definitions are contains within the <ssl-context-rules /> element.

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration>
    <authentication-client xmlns="urn:elytron:1.0">
        <ssl-context-rules>
            <rule use-ssl-context="...">
                ...
            </rule>
        </ssl-context-rules>
       ...
    </authentication-client>
</configuration>
```

Overall this means that authentication configuration matching is independent of SSLContext matching. By separating the rules from the configurations is means multiple rules can be defined that match against the same configuration.

The rules applied so first match wins and not most specific match wins, to achieve a most specific match wins configuration place the most specific rules at the beginning leaving the more general matches towards the end.

For both the <authentication-rules /> and the <ssl-context-rules /> the structure of the rules is identical other than one references an authentication configuration and the other references an SSLContext.

```
<rule use-configuration|use-ssl-context="...">
    <!-- At most one of the following two can be defined. -->
    <match-no-user />
    <match-user name="..." />
    <!-- Each of the following can be defined at most once. -->
    <match-protocol name="..." />
    <match-host name="..." />
    <match-path name="..." />
    <match-port number="..." />
    <match-urn name="..." />
    <match-domain name="..." />
    <match-abstract-type name="..." authority="..." />
</rule>
```

Where multiple matches are defined within a rule they must all match for the rule to apply. If a rule is defined with no match elements then it becomes a match all rule and will match anything, these can be useful at the end of the configuration to ensure something matches.

The individual match elements are: -

- **<match-no-user />** - user-info can be embedded within a URI, this rule matches when there is no user-info.
- *<match-user name="..." /> - Matches when the user-info embedded in the URI matches the name specified within this element.
- *<match-protocol name="..." /> - Matches the protocol within the URI against the name specified in this match element.
- *<match-host-name name="..." /> - Matches the host name from within the URI against the name specified in this match element.
- *<match-path name="..." /> - Matches the path from the URI against the name specified in this match element.
- *<match-port number="..." /> - Matches the port number specified within the URI against the number in this match element. This only matches against the number specified within the URI and not against any default derrived from the protocol.
- *<match-urn name="..." />" - Matches the scheme specific part of the URI against the name specified within this element.
- <match-domain-name name="..."/>* - Matches where the protocol of the URI is 'domain' and the scheme specific part of the URI is the name specified within this match element.
- **<match-abstract-type name="..." authority="..." />** - Matches the abstract type and/or authority against the values specified within this match element.

<authentication-configurations />

The <authentication-configurations /> element contains named configurations that can then be matched from the <authentication-rules />

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration>
    <authentication-client xmlns="urn:elytron:1.0">
        <authentication-configurations>
            <configuration name="...">
                <!-- Destination Overrides. -->
                <set-host name="..." />
                <set-port number="..." />
                <set-protocol name="..." />
                <!-- At most one of the following two elements. -->
                <set-user-name name="..." />
                <set-anonymous />
                <set-mechanism-realm name="..." />
                <rewrite-user-name-regex pattern="..." replacement="..." />
                <sasl-mechanism-selector selector="..." />
                <set-mechanism-properties>
                   <property key="..." value="..." />
                </set-mechanism-properties>
                <credentials>...</credentials>
                <set-authorization-name name="..." />
                <providers>...</providers>
                <!-- At most one of the following two elements. -->
                <use-provider-sasl-factory />
                <use-service-loader-sasl-factory module-name="..." />
            </configuration>
        </authentication-configurations>
    </authentication-client>
</configuration>
```

The elements within the <configuration /> element provide the following features: -

The first three elements override the destination.

- **<set-host-name name="..." />** - Override the host name for the authenticated call.
- **<set-port-number number="..." />** - Override the port number for the authenticated call.
- **<set-protocol name="..."/>** - Override the protocol for the authenticated call.

The next two are mutually exclusive and can be used to set the name for authentication or switch to anonymous authentication.

- **<set-user-name name="..."/>** - Set the user name to use for authentication.
- **<set-anonymous />** - Switch to anonymous authentication.

- **<set-mechanism-realm-name name="..." />** - Specify the name of the realm that will be selected by the SASL mechanism if required.
- **<rewrite-user-name-regex pattern="..." replacement="..." />** - A regular expression pattern and replacement to re-write the user name used for authentication.
- **<sasl-mechanism-selector selector="..." />** - A SASL mechanism selector using the syntax from org.wildfly.security.sasl.SaslMechanismSelector,fromString()
- **<set-mechanism-properties>...</set-mechanism-properties>** - One or more properties defined as <property key="..." value="..." /> to be passed to the authentication mechanisms.
- **<credentials>...</credentials>** - One or more credentials available for use during authentication.

The content of this element is the same as documented for <protection-parameter-credentials />

```
<credentials>
    <key-store-reference>...</key-store-reference>
    <credential-store-reference store="..." alias="..." clear-text="..." />
    <clear-password password="..." />
    <key-pair public-key-pem="..." private-key-pem="..." />
    <certificate private-key-pem="..." pem="..." />
    <public-key-pem>...</public-key-pem>
    <bearer-token value="..." />
    <oauth2-bearer-token token-endpoint-uri="...">...</oauth2-bearer-token>
</credentials>
```

- **<set-authorization-name name="..." />** - Specify the name that should be used for authorization if different from the authentication identity.
- **<providers/>** - This element is described in more detail within <providers /> and overrides the default or inherited provider discovery with a definition specific to this authentication configuration definition.

The final two elements are mutually exclusive and define how the SASL mechanism factories will be discovered for authentication.

- **<use-provider-sasl-factory />** - The java.security.Provider instances either inherited or defined in this configuration will be used to locate the available SASL client factories.
- **<use-service-loader-sasl-factory module-name="..." />** - SASL client factories will be discovered using service loader discovery on the specified module or if not specified using the ClassLoader loading the configuration.

<net-authenticator />

This element contains no specific configuration, however if present the org.wildfly.security.auth.util.ElytronAuthenticator will be registered with java.net.Authenticator.setDefault(Authenticator) meaning that the WildFly Elytron authentication client configuration can be used for authentication where the JDK APIs are used for HTTP calls which require authentication.

There are some limitations within this integration as the JDK will cache the authentication JVM wide from the first call so is better used in stand alone processes that don't require different credentials for different calls to the same URI,

<ssl-contexts />

The <ssl-contexts /> element holds individual names SSLContext definitions that can subsequently be matched by the <ssl-context-rules />.

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration>
    <authentication-client xmlns="urn:elytron:1.0">
        <ssl-contexts>
            <default-ssl-context name="..."/>
            <ssl-context name="...">
                <key-store-ssl-certificate>...</key-store-ssl-certificate>
                <trust-store key-store-name="..." />
                <cipher-suite selector="..." />
                <protocol names="... ..." />
                <provider-name name="..." />
                <providers>...</providers>
                <certificate-revocation-list path="..." maximum-cert-path="..." />
            </ssl-context>
        </ssl-contexts>
    </authentication-client>
</configuration>
```

The element <default-ssl-context name="..." /> simply takes the SSLContext obtainable from javax.net.ssl.SSLContext.getDefault() and assigns it a name so it can referenced from the <ssl-context-rules />. This element can be repeated meaning the default SSLContext can be referenced using different names.

The element <ssl-context /> is used to define a named configured SSLContext, each of the child elements is optional and can be specified at most once to build up the configuration of the SSLContext.

- **<key-store-ssl-certificate>** - Defines a reference to an entry within a KeyStore for the key and certificate to use in this SSLContext.

The overall structure of this element is: -

```
<key-store-ssl-certificate key-store-name="..." alias="...">
   <key-store-clear-password password="..." />
    <credential-store-reference store="..." alias="..." clear-text="..." />
   <key-store-credential>...</key-store-credential>
</key-store-ssl-certificate>
```

This structure is identical to the structure use in <key-store-credential />, the only difference being it is now to obtain the entry for the key and certificate, the nested elements however remain the protection parameter to unlock the entry.

- **<trust-store-key-store-name />** - A reference to a KeyStore that will be used to initialise the TrustManager.
- **<cipher-suite-selector />** - Configuration to filter the enabled cipher suites, the format of the selector is org.wildfly.security.ssl.CipherSuiteSelector.fromString(selector).

The following would be a cipher suite selector performing the default filtering.

```
<cipher-suite selector="DEFAULT" />
```

- **<protocol />** - used to define a space separated list of the protocols to be supported.
- **<provider-name />** - Once the available providers have been identified only the provider with the name defined on this element will be used.
- **<providers/>** - This element is described in more detail within <providers /> and overrides the default or inherited provider discovery with a definition specific to this SSLContext definition.
- **<certificate-revocation-list />** - The presence of this element enabled checking the peer's certificate against a certificate revocation list, this element defines both a path to the certificate revocation list and also specifies the maximum number of non-self-issued intermediate certificates that may exist in a certification path

<providers />

The *<providers />* element is used to define how java.security.Provider instances are located when required. The other configuration sections of *<authentication-client />* are independent of each other, the *<providers />* configuration however applies to the current element and it's children unless overridden, this configuration can be specified in the following locations.

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration>
    <authentication-client xmlns="urn:elytron:1.0">
        <providers />
        ...
        <credential-stores>
            <credential-store name="..">
                ...
                <providers />
            </credential-store>
        <credential-stores>
        ...
        <authentication-configurations>
            <authentication-configuration name="...">
                ...
                <providers />
            </authentication-configuration>
        </authentication-configurations>
        ...
        <ssl-contexts>
            <ssl-context name="...">
                ...
                <providers />
            </ssl-context>
        </ssl-contexts>
    </authentication-client>
</configuration>
```

If an individual <credential-store />, <authentication-configuration />, or <ssl-context /> contains a <providers /> definition that that definition will apply specifically to that instance. If a configured item does not contain a <providers /> definition but a top level <providers /> is defined within <authentication-configuration /> then that will be used instead.

The <providers /> element can be defined as: -

```
<providers>
    <global />
    <use-service-loader module-name="..." />
</providers>
```

Both the child elements are optional, can appear in any order and can be repeated although repeating <global /> would not really be beneficial.

- **<global />** - The providers from java.security.Security.getProviders()
- **<credential-stores />** - Providers loaded using service loader discovery from the module specified, if no module is specified the ClassLoader which loaded the authentication client is used.

Where no <provider /> configuration exists the default behaviour is the equivalent of: -

```
<providers>
    <use-service-loader />
    <global />
</providers>
```

This gives the WildFly Elytron Provider priority over any globally registered Providers but also allows for the globally registered providers to be used.

16.2.2 <jboss-ejb-client /> - EJB Client

The *<jboss-ejb-client />* element in a wildfly-config.xml file can be used to specify EJB Client configuration. This element is from the “urn:jboss:wildfly-client-ejb:3.0” namespace, e.g.

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration>
...
    <jboss-ejb-client xmlns="urn:jboss:wildfly-client-ejb:3.0">
        ...
    </jboss-ejb-client>
...
</configuration>
```

This section describes the child elements and attributes that can be configured within this element.

The *<jboss-ejb-client />* element can optionally contain the following three child elements, as described in the next sections:

- **<invocation-timeout />**
- **<global-interceptors />**
- **<connections />**

<invocation-timeout />

This element is used to specify an EJB invocation timeout. It has one attribute which is required:

Attribute	Description
seconds	The timeout, in seconds, for the EJB handshake or method invocation request/response cycle. The invocation of any method throws a `java.util.concurrent.TimeoutException` if the execution takes longer than the timeout period. The server side will not be interrupted.

<global-interceptors />

This element is used to specify global EJB client interceptors. It can contain any number of *<interceptor />* elements.

<interceptor />

This element is used to specify an EJB client interceptor. It has two attributes:

Attribute	Description
class	The name of a class that implements the `org.jboss.ejb.client.EJBClientInterceptor` interface.
module	The optional name of the module that should be used to load the interceptor class.

<connections />

This element is used to specify EJB client connections. It can contain any number of *<connection />* elements.

<connection />

This element is used to specify an EJB client connection. It has one required attribute. It can also optionally contain an *<interceptors />* element.

Attribute	Description
uri	The connection destination URI.

<interceptors />

This element is used to specify EJB client interceptors and can contain any number of *<interceptor />* elements.

16.2.3 <endpoint /> - Remoting Client

You can use the `endpoint` element, which is in the `urn:jboss-remoting:5.0` namespace, to configure a JBoss Remoting client endpoint using the `wildfly-config.xml` file. This section describes how to configure a JBoss Remoting client using this element.

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration>
...
    <endpoint xmlns="urn:jboss-remoting:5.0">
       ...
    </endpoint>
...
</configuration>
```

This section describes the child elements and attributes that can be configured within this element.

The <*endpoint* /> element contains the following optional attribute:

Attribute Name	Attribute Description
`name`	The endpoint name. If not given, an endpoint name will be derived from the system's host name, if possible.

The <*endpoint* /> element can optionally contain the following two child elements, as described in the next sections:

- **<providers />**
- **<connections />**

The configured endpoint will use the default XNIO configuration.

<providers />

This optional element specifies transport providers for the remote endpoint. It can contain any number of <*provider* /> sub-elements.

<provider />

This element defines a remote transport provider provider. It has the following attributes.

Attribute Name	Attribute Description
`scheme`	The primary URI scheme which corresponds to this provider. This attribute is required.
`aliases`	A space-separated list of other URI scheme names that are also recognized for this provider . This attribute is optional.
`module`	The name of the module that contains the provider implementation. This attribute is optional; if not given, the class loader of JBoss Remoting itself will be searched for the provider class.
`class`	The name of the class that implements the transport provider. This attribute is optional; if not given, the Java `java.util.ServiceLoader` facility will be used to search for the provider class.

This element has no content.

<connections />

This optional element specifies connections for the remote endpoint. It can contain any number of connection elements.

<connection />

This element defines a connection for the remote endpoint. It has the following attributes.

Attribute Name	Attribute Description
`destination`	The destination URI for the connection. This attribute is required.
`read-timeout`	The timeout, in seconds, for read operations on the corresponding socket. This attribute is optional, however it should only be given if a `heartbeat-interval` is defined.
`write-timeout`	The timeout, in seconds, for a write operation. This attribute is optional, however it should only be given if a `heartbeat-interval` is defined..
`ip-traffic-class`	Defines the numeric IP traffic class to use for this connection's traffic. This attribute is optional.
`tcp-keepalive`	Boolean setting that determines whether to use TCP keepalive. This attribute is optional.
`heartbeat-interval`	The interval, in milliseconds, to use when checking for a connection heartbeat. This attribute is optional.

Example Remoting Client Configuration in the wildfly-config.xml File

```
<configuration>
...
    <endpoint xmlns="urn:jboss-remoting:5.0">
        <connections>
            <connection destination="remote+http://10.20.30.40:8080" read-timeout="50"
write-timeout="50" heartbeat-interval="10000"/>
        </connections>
    </endpoint>
...
</configuration>
```

16.2.4 <worker /> - XNIO Client

You can use the `worker` element, which is in the `urn:xnio:3.5` namespace, to configure a default XNIO worker using the `wildfly-config.xml` file. This section describes how to do this.

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration>
...
    <worker xmlns="urn:xnio:3.5">
       ...
    </worker>
...
</configuration>
```

This section describes the child elements that can be configured within this root `worker` element.

The *<worker />* element can optionally contain the following child elements, as described in the next sections:

- **<daemon-threads />**
- **<worker-name />**
- **<pool-size />**
- **<task-keepalive />**
- **<io-threads />**
- **<stack-size />**
- **<outbound-bind-addresses />**

<daemon-threads />

This optional element takes a single required attribute:

Attribute Name	Attribute Description
`value`	The value of the setting (required). A value of `true` indicates that worker and task threads should be daemon threads, and `false` indicates that they should not be daemon threads. If this element is not given, a value of `true` is assumed.

This element has no content.

<worker-name />

This element defines the name of the worker. The worker name will appear in thread dumps and in JMX.

Attribute Name	Attribute Description
`value`	The worker's name (required).

This element has no content.

<pool-size />

This optional element defines the size parameters of the worker's task thread pool. The following attributes are allowed:

Attribute Name	Attribute Description
`max-threads`	A positive integer which specifies the maximum number of threads that should be created (required).

<task-keepalive />

This optional element establishes the keep-alive time of task threads before they may be expired.

Attribute Name	Attribute Description
`value`	A positive integer which represents the minimum number of seconds to keep idle threads alive (required).

<io-threads />

This optional element determines how many I/O (selector) threads should be maintained. Generally this number should be a small constant multiple of the number of available cores.

Attribute Name	Attribute Description
`value`	A positive integer value for the number of I/O threads (required).

<stack-size />

This optional element establishes the desired minimum thread stack size for worker threads.

Attribute Name	Attribute Description
`value`	A positive integer value which indicates the requested stack size, in bytes (required).

<outbound-bind-addresses />

This optional element specifies bind addresses to use for outbound connections. Each bind address mapping consists of a destination IP address block, and a bind address and optional port number to use for connections to destinations within that block.

<bind-address />

This element defines an individual bind address mapping.

Attribute Name	Attribute Description
`match`	The IP address block in CIDR notation to match (required).
`bind-address`	The IP address to bind to if the address block matches (required).
`bind-port`	A specific port number to bind to if the address block matches (optional, defaults to 0 meaning "any port").

16.3 <authentication-client /> - WildFly Elytron

The *<authentication-client/>* element can be added to the wildfly-config.xml configuration to define configuration in relation to authentication configuration for outbound connections and SSL configuration for outbound connections e.g.

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration>
    <authentication-client xmlns="urn:elytron:1.0">
       ...
    </authentication-client>
</configuration>
```

Note: A single wildfly-config.xml could be used by multiple projects using multiple versions of Wildfly Elytron, newer versions of WildFly Elytron will introduce new configuration using later namespace versions however they will still continue to support the previously released schemas until advertised otherwise. For the configuration to be compatible with this either use a schema and namespace compatible with all the versions in use, or multiple authentication-client elements can be added, these should be added ordered by namespace youngest to oldest. If a configuration with a later namespace is discovered by a newer WildFly Elytron client it will be used and parsing will not look for an older version as well.

The *<authentication-client />* configuration can contain the following sections: -

- **<credential-stores />** - Definitions of credential stores to be referenced from elsewhere in the configuration.
- **<key-stores />** - Definitions of KeyStores to be referenced elsewhere in the configuration.
- **<authentication-rules />** - Rules to be applied for outbound connections to match against an appropriate authentication configuration.
- **<authentication-configurations />** - The individual authentication configurations that will be matched by the authentication rules.
- **<net-authenticator />** - Flag to enable integration with the java.net.Authenticator.
- **<ssl-context-rules />** - Rules to be applied for outbound connections to match against an appropriate SSL context configuration.
- **<ssl-contexts />** - Individual SSL context definitions that will be matched by the ssl context rules.
- **<providers/>** - Definition of how java.security.Provider instances will be discovered.

16.3.1 <credential-stores />

The <credential-stores /> element can be used to define individual named credential stores that can subsequently be used elsewhere in the configuration to reference stored credentials as an alternative to the credentials being embedded in the configuration.

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration>
    <authentication-client xmlns="urn:elytron:1.0">
        <credential-stores>
            <credential-store name="..." type="..." provider="..." >
                <attributes>
                    <attribute name="..." value="..." />
                </attributes>
                <protection-parameter-credentials>...</protection-parameter-credentials>
            </credential-store>
        </credential-stores>
    </authentication-client>
</configuration>
```

In addition to the name an individual <credential-store /> definition can contain the following optional attributes: -

1. **type** - The type of the credential store, e.g.KeyStoreCredentialStore.
2. **provider** - The name of the java.security.Provider to use to load the credential store.

The following child elements can also be added to configure the credential store.

- **<attributes .>** - Definition of configuration attributes used to initialise the credential store.

e.g.

```
<attributes>
    <attribute name="..." value="..." />
</attributes>
```

The <attribute/> element can be repeated as many times as is required for the configuration.

- **<protection-parameter-credentials />** - One or more credentials to be assembled into a protection parameter when initialising the credential store.

The <protection-paramter-credentials /> element can contain one more more child elements, which of these are actually supported will depend on the credential store implementation: -

```
<protection-parameter-credentials>
    <key-store-reference>...</key-store-reference>
    <credential-store-reference store="..." alias="..." clear-text="..." />
    <clear-password password="..." />
    <key-pair public-key-pem="..." private-key-pem="..." />
    <certificate private-key-pem="..." pem="..." />
    <public-key-pem>...</public-key-pem>
    <bearer-token value="..." />
    <oauth2-bearer-token token-endpoint-uri="...">...</oauth2-bearer-token>
</protection-parameter-credentials>
```

The potential child elements of <protection-parameter-credentials /> are: -

- **<key-store-reference>** - Defines a reference to an entry within a KeyStore for an entry to use.

The overall structure of this element is: -

```
<key-store-reference key-store-name="..." alias="...">
   <key-store-clear-password password="..." />
   <credential-store-reference store="..." alias="..." clear-text="..." />
   <key-store-credential>...</key-store-credential>
</key-store-reference>
```

This structure is identical to the structure use in <key-store-credential />.

- **<credential-store-reference store="..." alias="..." clear-text="..." />** - Reference to a credential stored in a credential store.
- **<clear-password password="..." />** - A password specified in the clear.
- **<key-pair public-key-pem="..." private-key-pem="..." />** - A public and private key pair.
- <certificate private-key-pem="..." pem="..." />* - A pem encoded private key and corresponding certificate.
- **<public-key-pem>...</public-key-pem>** - A pem encoded public key.

- <bearer-token value="..." />* - A bearer token
- **<oauth2-bearer-token>...</oauth2-bearer-token>** - An oath2 bearer token.

The full structure of this element is: -

```
<oauth2-bearer-token token-endpoint-uri="...">
    <client-credentials client-id="..." client-secret="..." />
    <resource-owner-credentials name="..." password="..." />
</oauth2-bearer-token>
```

16.3.2 <key-stores />

The <key-stores /> element can be used to define individual key-store definitions that can subsequently be referenced from alternative locations within the configuration.

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration>
    <authentication-client xmlns="urn:elytron:1.0">
        <key-stores>
            <key-store name="...">
               <!-- One of the following to specify where to load the KeyStore from. -->
               <file-name name="..." />
               <load-from uri-"..." />
               <resource name="..." />
               <!-- One of the following to specify the protection parameter to unlock the
KeyStore. -->
               <key-store-clear-password password="..." />
               <key-store-credential>...</key-store-credential>
            </key-store>
        </key-stores>
       ...
    </authentication-client>
</configuration>
```

An individual <key-store /> definition must contain exactly one of the following elements to define where to load the store from.

1. **<file name="..." />** - Load from file where 'name' is the name of the file.
2. **<load-from uri="..." />** - Load the file from the URI specified.
3. **<resource name="..." />** - Load as a resource from the Thread context classloader where 'name' is the name of the resource to load.

Exactly one of the following elements must also be present to specify the protection parameter for initialisation of the KeyStore.

1. **<key-store-clear-password password="..." />** - A password specified in the clear.
2. **<key-store-credential>...</key-store-credential>** - A reference to another KeyStore to obtain an Entry to use as the protection parameter to access this KeyStore.

The structure of the <key-store-credential /> element is.

```
<key-store-credential key-store-name="..." alias="...">
   <key-store-clear-password password="..." />
   <credential-store-reference store="..." alias="..." clear-text="..." />
   <key-store-credential>...</key-store-credential>
</key-store-credential>
```

This element contains two attributes: -

1. **key-store-name** *(Mandatory)* - Name of the KeyStore being referenced to load the entry from.
2. **alias** *(Optional)* - The alias of the entry to load from the referenced KeyStore, this can only be omitted for KeyStores that contain only a single entry.

Java KeyStores also make use of a protection parameter when accessing a single entry in addition to the protection parameter to load the KeyStore, exactly one of the following elements must be present to specify the protection parameter of the entry being loaded.

1. **<key-store-clear-password password="..." />** - A password specified in the clear.
2. **<credential-store-reference store="..." alias="..." clear-text="..." />** - Reference to a credential stored in a credential store.
3. **<key-store-credential>...</key-store-credential>** - A reference to another KeyStore to obtain an Entry to use as the protection parameter to access the alias.

The <key-store-credential /> is exactly the same, this means theoretically a chain of references could be used to lead to the unlocking of the required alias.

16.3.3 <authentication-rules /> and <ssl-context-rules />

When either an authentication-configuration or an ssl-context is required the URI of the resources being accessed as well as an optional abstract type and abstract type authority and matched against the rules defined in the configuration to identify which authentication-configuration or ssl-context should be used.

The rules to match <authentication-configuration /> instances are defined within the <authentication-rules /> element.

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration>
    <authentication-client xmlns="urn:elytron:1.0">
        <authentication-rules>
            <rule use-configuration="...">
                ...
            </rule>
        </authentication-rules>
       ...
    </authentication-client>
</configuration>
```

The rules to match against the <ssl-context /> definitions are contains within the <ssl-context-rules /> element.

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration>
    <authentication-client xmlns="urn:elytron:1.0">
        <ssl-context-rules>
            <rule use-ssl-context="...">
                ...
            </rule>
        </ssl-context-rules>
       ...
    </authentication-client>
</configuration>
```

Overall this means that authentication configuration matching is independent of SSLContext matching. By separating the rules from the configurations is means multiple rules can be defined that match against the same configuration.

The rules applied so first match wins and not most specific match wins, to achieve a most specific match wins configuration place the most specific rules at the beginning leaving the more general matches towards the end.

For both the <authentication-rules /> and the <ssl-context-rules /> the structure of the rules is identical other than one references an authentication configuration and the other references an SSLContext.

```
<rule use-configuration|use-ssl-context="...">
    <!-- At most one of the following two can be defined. -->
    <match-no-user />
    <match-user name="..." />
    <!-- Each of the following can be defined at most once. -->
    <match-protocol name="..." />
    <match-host name="..." />
    <match-path name="..." />
    <match-port number="..." />
    <match-urn name="..." />
    <match-domain name="..." />
    <match-abstract-type name="..." authority="..." />
</rule>
```

Where multiple matches are defined within a rule they must all match for the rule to apply. If a rule is defined with no match elements then it becomes a match all rule and will match anything, these can be useful at the end of the configuration to ensure something matches.

The individual match elements are: -

- **<match-no-user />** - user-info can be embedded within a URI, this rule matches when there is no user-info.
- *<match-user name="..." /> - Matches when the user-info embedded in the URI matches the name specified within this element.
- *<match-protocol name="..." /> - Matches the protocol within the URI against the name specified in this match element.
- *<match-host-name name="..." /> - Matches the host name from within the URI against the name specified in this match element.
- *<match-path name="..." /> - Matches the path from the URI against the name specified in this match element.
- *<match-port number="..." /> - Matches the port number specified within the URI against the number in this match element. This only matches against the number specified within the URI and not against any default derrived from the protocol.
- *<match-urn name="..." />" - Matches the scheme specific part of the URI against the name specified within this element.
- <match-domain-name name="..."/>* - Matches where the protocol of the URI is 'domain' and the scheme specific part of the URI is the name specified within this match element.
- **<match-abstract-type name="..." authority="..." />** - Matches the abstract type and/or authority against the values specified within this match element.

16.3.4 <authentication-configurations />

The <authentication-configurations /> element contains named configurations that can then be matched from the <authentication-rules />

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration>
    <authentication-client xmlns="urn:elytron:1.0">
        <authentication-configurations>
            <configuration name="...">
                <!-- Destination Overrides. -->
                <set-host name="..." />
                <set-port number="..." />
                <set-protocol name="..." />
                <!-- At most one of the following two elements. -->
                <set-user-name name="..." />
                <set-anonymous />
                <set-mechanism-realm name="..." />
                <rewrite-user-name-regex pattern="..." replacement="..." />
                <sasl-mechanism-selector selector="..." />
                <set-mechanism-properties>
                   <property key="..." value="..." />
                </set-mechanism-properties>
                <credentials>...</credentials>
                <set-authorization-name name="..." />
                <providers>...</providers>
                <!-- At most one of the following two elements. -->
                <use-provider-sasl-factory />
                <use-service-loader-sasl-factory module-name="..." />
            </configuration>
        </authentication-configurations>
    </authentication-client>
</configuration>
```

The elements within the <configuration /> element provide the following features: -

The first three elements override the destination.

- **<set-host-name name="..." />** - Override the host name for the authenticated call.
- **<set-port-number number="..." />** - Override the port number for the authenticated call.
- **<set-protocol name="..."/>** - Override the protocol for the authenticated call.

The next two are mutually exclusive and can be used to set the name for authentication or switch to anonymous authentication.

- **<set-user-name name="..."/>** - Set the user name to use for authentication.
- **<set-anonymous />** - Switch to anonymous authentication.

- **<set-mechanism-realm-name name="..." />** - Specify the name of the realm that will be selected by the SASL mechanism if required.
- **<rewrite-user-name-regex pattern="..." replacement="..." />** - A regular expression pattern and replacement to re-write the user name used for authentication.
- **<sasl-mechanism-selector selector="..." />** - A SASL mechanism selector using the syntax from org.wildfly.security.sasl.SaslMechanismSelector,fromString()
- **<set-mechanism-properties>...</set-mechanism-properties>** - One or more properties defined as <property key="..." value="..." /> to be passed to the authentication mechanisms.
- **<credentials>...</credentials>** - One or more credentials available for use during authentication.

The content of this element is the same as documented for <protection-parameter-credentials />

```
<credentials>
    <key-store-reference>...</key-store-reference>
    <credential-store-reference store="..." alias="..." clear-text="..." />
    <clear-password password="..." />
    <key-pair public-key-pem="..." private-key-pem="..." />
    <certificate private-key-pem="..." pem="..." />
    <public-key-pem>...</public-key-pem>
    <bearer-token value="..." />
    <oauth2-bearer-token token-endpoint-uri="...">...</oauth2-bearer-token>
</credentials>
```

- **<set-authorization-name name="..." />** - Specify the name that should be used for authorization if different from the authentication identity.
- **<providers/>** - This element is described in more detail within <providers /> and overrides the default or inherited provider discovery with a definition specific to this authentication configuration definition.

The final two elements are mutually exclusive and define how the SASL mechanism factories will be discovered for authentication.

- **<use-provider-sasl-factory />** - The java.security.Provider instances either inherited or defined in this configuration will be used to locate the available SASL client factories.
- **<use-service-loader-sasl-factory module-name="..." />** - SASL client factories will be discovered using service loader discovery on the specified module or if not specified using the ClassLoader loading the configuration.

16.3.5 <net-authenticator />

This element contains no specific configuration, however if present the org.wildfly.security.auth.util.ElytronAuthenticator will be registered with java.net.Authenticator.setDefault(Authenticator) meaning that the WildFly Elytron authentication client configuration can be used for authentication where the JDK APIs are used for HTTP calls which require authentication.

There are some limitations within this integration as the JDK will cache the authentication JVM wide from the first call so is better used in stand alone processes that don't require different credentials for different calls to the same URI,

16.3.6 <ssl-contexts />

The <ssl-contexts /> element holds individual names SSLContext definitions that can subsequently be matched by the <ssl-context-rules />.

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration>
    <authentication-client xmlns="urn:elytron:1.0">
        <ssl-contexts>
            <default-ssl-context name="..."/>
            <ssl-context name="...">
                <key-store-ssl-certificate>...</key-store-ssl-certificate>
                <trust-store key-store-name="..." />
                <cipher-suite selector="..." />
                <protocol names="... ..." />
                <provider-name name="..." />
                <providers>...</providers>
                <certificate-revocation-list path="..." maximum-cert-path="..." />
            </ssl-context>
        </ssl-contexts>
    </authentication-client>
</configuration>
```

The element <default-ssl-context name="..." /> simply takes the SSLContext obtainable from javax.net.ssl.SSLContext.getDefault() and assigns it a name so it can referenced from the <ssl-context-rules />. This element can be repeated meaning the default SSLContext can be referenced using different names.

The element <ssl-context /> is used to define a named configured SSLContext, each of the child elements is optional and can be specified at most once to build up the configuration of the SSLContext.

- **<key-store-ssl-certificate>** - Defines a reference to an entry within a KeyStore for the key and certificate to use in this SSLContext.

The overall structure of this element is: -

```
<key-store-ssl-certificate key-store-name="..." alias="...">
   <key-store-clear-password password="..." />
    <credential-store-reference store="..." alias="..." clear-text="..." />
   <key-store-credential>...</key-store-credential>
</key-store-ssl-certificate>
```

This structure is identical to the structure use in <key-store-credential />, the only difference being it is now to obtain the entry for the key and certificate, the nested elements however remain the protection parameter to unlock the entry.

- **<trust-store-key-store-name />** - A reference to a KeyStore that will be used to initialise the TrustManager.
- **<cipher-suite-selector />** - Configuration to filter the enabled cipher suites, the format of the selector is org.wildfly.security.ssl.CipherSuiteSelector.fromString(selector).

The following would be a cipher suite selector performing the default filtering.

```
<cipher-suite selector="DEFAULT" />
```

- **<protocol />** - used to define a space separated list of the protocols to be supported.
- **<provider-name />** - Once the available providers have been identified only the provider with the name defined on this element will be used.
- **<providers/>** - This element is described in more detail within <providers /> and overrides the default or inherited provider discovery with a definition specific to this SSLContext definition.
- **<certificate-revocation-list />** - The presence of this element enabled checking the peer's certificate against a certificate revocation list, this element defines both a path to the certificate revocation list and also specifies the maximum number of non-self-issued intermediate certificates that may exist in a certification path

16.3.7 <providers />

The *<providers />* element is used to define how java.security.Provider instances are located when required. The other configuration sections of *<authentication-client />* are independent of each other, the *<providers />* configuration however applies to the current element and it's children unless overridden, this configuration can be specified in the following locations.

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration>
    <authentication-client xmlns="urn:elytron:1.0">
        <providers />
        ...
        <credential-stores>
            <credential-store name="..">
                ...
                <providers />
            </credential-store>
        <credential-stores>
        ...
        <authentication-configurations>
            <authentication-configuration name="...">
                ...
                <providers />
            </authentication-configuration>
        </authentication-configurations>
        ...
        <ssl-contexts>
            <ssl-context name="...">
                ...
                <providers />
            </ssl-context>
        </ssl-contexts>
    </authentication-client>
</configuration>
```

If an individual <credential-store />, <authentication-configuration />, or <ssl-context /> contains a <providers /> definition that that definition will apply specifically to that instance. If a configured item does not contain a <providers /> definition but a top level <providers /> is defined within <authentication-configuration /> then that will be used instead.

The <providers /> element can be defined as: -

```
<providers>
    <global />
    <use-service-loader module-name="..." />
</providers>
```

Both the child elements are optional, can appear in any order and can be repeated although repeating <global /> would not really be beneficial.

- **<global />** - The providers from java.security.Security.getProviders()
- **<credential-stores />** - Providers loaded using service loader discovery from the module specified, if no module is specified the ClassLoader which loaded the authentication client is used.

Where no <provider /> configuration exists the default behaviour is the equivalent of: -

```
<providers>
    <use-service-loader />
    <global />
</providers>
```

This gives the WildFly Elytron Provider priority over any globally registered Providers but also allows for the globally registered providers to be used.

16.4 <jboss-ejb-client /> - EJB Client

The *<jboss-ejb-client />* element in a wildfly-config.xml file can be used to specify EJB Client configuration. This element is from the “urn:jboss:wildfly-client-ejb:3.0” namespace, e.g.

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration>
...
    <jboss-ejb-client xmlns="urn:jboss:wildfly-client-ejb:3.0">
       ...
    </jboss-ejb-client>
...
</configuration>
```

This section describes the child elements and attributes that can be configured within this element.

The *<jboss-ejb-client />* element can optionally contain the following three child elements, as described in the next sections:

- **<invocation-timeout />**
- **<global-interceptors />**
- **<connections />**

16.4.1 <invocation-timeout />

This element is used to specify an EJB invocation timeout. It has one attribute which is required:

Attribute	Description
seconds	The timeout, in seconds, for the EJB handshake or method invocation request/response cycle. The invocation of any method throws a `java.util.concurrent.TimeoutException` if the execution takes longer than the timeout period. The server side will not be interrupted.

16.4.2 <global-interceptors />

This element is used to specify global EJB client interceptors. It can contain any number of *<interceptor />* elements.

16.4.3 <interceptor />

This element is used to specify an EJB client interceptor. It has two attributes:

Attribute	Description
class	The name of a class that implements the `org.jboss.ejb.client.EJBClientInterceptor` interface.
module	The optional name of the module that should be used to load the interceptor class.

16.4.4 <connections />

This element is used to specify EJB client connections. It can contain any number of *<connection />* elements.

16.4.5 <connection />

This element is used to specify an EJB client connection. It has one required attribute. It can also optionally contain an *<interceptors />* element.

Attribute	Description
uri	The connection destination URI.

16.4.6 <interceptors />

This element is used to specify EJB client interceptors and can contain any number of *<interceptor />* elements.

16.5 <endpoint /> - Remoting Client

You can use the `endpoint` element, which is in the `urn:jboss-remoting:5.0` namespace, to configure a JBoss Remoting client endpoint using the `wildfly-config.xml` file. This section describes how to configure a JBoss Remoting client using this element.

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration>
...
    <endpoint xmlns="urn:jboss-remoting:5.0">
       ...
    </endpoint>
...
</configuration>
```

This section describes the child elements and attributes that can be configured within this element.

The *<endpoint />* element contains the following optional attribute:

Attribute Name	Attribute Description
name	The endpoint name. If not given, an endpoint name will be derived from the system's host name, if possible.

The *<endpoint />* element can optionally contain the following two child elements, as described in the next sections:

- **<providers />**
- **<connections />**

The configured endpoint will use the default XNIO configuration.

16.5.1 <providers />

This optional element specifies transport providers for the remote endpoint. It can contain any number of *<provider />* sub-elements.

<provider />

This element defines a remote transport provider provider. It has the following attributes.

Attribute Name	Attribute Description
`scheme`	The primary URI scheme which corresponds to this provider. This attribute is required.
`aliases`	A space-separated list of other URI scheme names that are also recognized for this provider . This attribute is optional.
`module`	The name of the module that contains the provider implementation. This attribute is optional; if not given, the class loader of JBoss Remoting itself will be searched for the provider class.
`class`	The name of the class that implements the transport provider. This attribute is optional; if not given, the Java `java.util.ServiceLoader` facility will be used to search for the provider class.

This element has no content.

16.5.2 <connections />

This optional element specifies connections for the remote endpoint. It can contain any number of connection elements.

<connection />

This element defines a connection for the remote endpoint. It has the following attributes.

Attribute Name	Attribute Description
`destination`	The destination URI for the connection. This attribute is required.
`read-timeout`	The timeout, in seconds, for read operations on the corresponding socket. This attribute is optional, however it should only be given if a `heartbeat-interval` is defined.
`write-timeout`	The timeout, in seconds, for a write operation. This attribute is optional, however it should only be given if a `heartbeat-interval` is defined..
`ip-traffic-class`	Defines the numeric IP traffic class to use for this connection's traffic. This attribute is optional.
`tcp-keepalive`	Boolean setting that determines whether to use TCP keepalive. This attribute is optional.
`heartbeat-interval`	The interval, in milliseconds, to use when checking for a connection heartbeat. This attribute is optional.

16.5.3 Example Remoting Client Configuration in the wildfly-config.xml File

```
<configuration>
...
    <endpoint xmlns="urn:jboss-remoting:5.0">
        <connections>
            <connection destination="remote+http://10.20.30.40:8080" read-timeout="50"
write-timeout="50" heartbeat-interval="10000"/>
        </connections>
    </endpoint>
...
</configuration>
```

16.6 <worker /> - XNIO Client

You can use the `worker` element, which is in the `urn:xnio:3.5` namespace, to configure a default XNIO worker using the `wildfly-config.xml` file. This section describes how to do this.

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration>
...
    <worker xmlns="urn:xnio:3.5">
       ...
    </worker>
...
</configuration>
```

This section describes the child elements that can be configured within this root `worker` element.

The *<worker />* element can optionally contain the following child elements, as described in the next sections:

- **<daemon-threads />**
- **<worker-name />**
- **<pool-size />**
- **<task-keepalive />**
- **<io-threads />**
- **<stack-size />**
- **<outbound-bind-addresses />**

16.6.1 <daemon-threads />

This optional element takes a single required attribute:

Attribute Name	Attribute Description
`value`	The value of the setting (required). A value of `true` indicates that worker and task threads should be daemon threads, and `false` indicates that they should not be daemon threads. If this element is not given, a value of `true` is assumed.

This element has no content.

16.6.2 <worker-name />

This element defines the name of the worker. The worker name will appear in thread dumps and in JMX.

Attribute Name	Attribute Description
`value`	The worker's name (required).

This element has no content.

16.6.3 <pool-size />

This optional element defines the size parameters of the worker's task thread pool. The following attributes are allowed:

Attribute Name	Attribute Description
`max-threads`	A positive integer which specifies the maximum number of threads that should be created (required).

16.6.4 <task-keepalive />

This optional element establishes the keep-alive time of task threads before they may be expired.

Attribute Name	Attribute Description
`value`	A positive integer which represents the minimum number of seconds to keep idle threads alive (required).

16.6.5 <io-threads />

This optional element determines how many I/O (selector) threads should be maintained. Generally this number should be a small constant multiple of the number of available cores.

Attribute Name	Attribute Description
`value`	A positive integer value for the number of I/O threads (required).

16.6.6 <stack-size />

This optional element establishes the desired minimum thread stack size for worker threads.

Attribute Name	Attribute Description
`value`	A positive integer value which indicates the requested stack size, in bytes (required).

16.6.7 <outbound-bind-addresses />

This optional element specifies bind addresses to use for outbound connections. Each bind address mapping consists of a destination IP address block, and a bind address and optional port number to use for connections to destinations within that block.

<bind-address />

This element defines an individual bind address mapping.

Attribute Name	Attribute Description
`match`	The IP address block in CIDR notation to match (required).
`bind-address`	The IP address to bind to if the address block matches (required).
`bind-port`	A specific port number to bind to if the address block matches (optional, defaults to 0 meaning "any port").