CIPP Reference Guide

This documentation covers CIPP version 2.28.

http://www.dimedis.de/
http://www.perl.com/CPAN/modules/by-module/CIPP/

Table Of Contents

CHAPTER 1

CIPP - CGI Perl Preprocessor

This chapter gives a high-level overview about what CIPP is, where it can be used and introduces you to the general syntax of the language.

Introduction

The name CIPP is an acronym for CgI Perl Preprocessor. With CGI, a web server calls a program which generates a HTML page. The CGI allows passing of parameters, so the returned page might look different depending on the input to the program. This is what is commonly refered to as a „dynamic“ page.

CGI programms are just like normal ones, only there is a lot of code printing out HTML statements. The majority of the code is concerned about the layout of the generated page. This is a nuisance for two reasons: first, it is difficult to see the structure of the generated page by looking at the source code, second, a lot of the code just consists of „print“ statements - these are boring to write.

CIPP takes another approach to CGI programming: you basically write an ordinary HTML page and insert into the page the code, which is responsible for the dynamic parts. This way, you can easily see the structure of the page and for generating HTML, you can simply write the HTML directly onto the page.

CIPP generates Perl code

CIPP is a preprocessor which generates pure Perl code out of your CIPP embedded HTML pages. Depending on your environment, this Perl code can either be installed as a CGI program on the webserver or is executed immediately through an appropriate handler. More details about the different environments and their properties are discussed later in this document.

Here is a little example of a CIPP code snippet to demonstrate the simplicity of the preprocessing mechanism (this anticipates some basics of the CIPP programming language, a detailed description of the language follow beyond this chapter).

```
<?IF COND="$event eq 'show'">
  The value of the variable 'foo' is:<BR>
  <B>$foo</B>
<?/IF>
```

You will get a HTML formatted content of the Perl variable $foo, assumed the variable $event contains the string 'show'.

CIPP will generate Perl code similar to this.

```
if ( $event eq 'show' ) {
  print "The value of the variable 'foo' is:<BR>\n";
  print "<B>$foo</B>\n";
}
```

This was really a simple example. The CIPP **<?IF>** is translated to a Perl 'if' command. The non CIPP text blocks (usually containing some HTML) are translated to a Perl 'print' command. There are many, more complex CIPP commands that save you a lot of work.

So, here you can see the difference between CIPP and ordinary CGI programming. With CIPP, HTML is normal and code is embedded in a way which almost looks like HTML. CGI programs, on the other hand, contain a lot of print statements which makes them hard to read.

Ok, message understood. Now you know what CIPP basically does for you. In the next chapter you will learn in what way and environment you can apply it.

Environments where CIPP can be used

As mentioned above there are three different environments where you can use CIPP programs:

- CIPP::CGI - using CIPP via a central CGI wrapper program
- Apache::CGI - using CIPP as a module inside the Apache webserver
- new.spirit - managing projects of many CIPP files, generating standalone CGI programs for production web systems.

A discussion of these three possible use cases follows, where the architecture of each environment is described briefly. There are extra chapters with configuration details about all of them.

CGI::CIPP

CGI::CIPP is a Perl module which enables you to use CIPP on every CGI capable webserver. It is based on a central wrapper script, which does all the preprocessing. It executes the generated Perl code directly afterwards. Additionally, it implements a filesystem based cache for the generated code. Preprocessing is done only when the corresponding CIPP source code changed on disk, otherwise this step is skipped.

CGI::CIPP is prepared for usage inside a persistent Perl environment, e.g. in conjunction with the CGI::SpeedyCGI module, which is not part of the CIPP distribution, but freely available on CPAN. CGI::CIPP will cache the Perl compiled programs as subroutines. Subsequent calls to the same CIPP page are answered immediately, because neither CIPP preprocessing nor Perl compiling needs to be done in this case.

Your CIPP source files are placed in a particular directory on a webserver. With some additional webserver configuration you can handle them as „normal“ HTML documents beneath other webserver documents like images or traditional static HTML documents. See the chapter about CGI::CIPP configuration for details.

Apache::CIPP

The architecture of the Apache::CIPP is very similar to the one of CGI::CIPP. The main difference is that the central CGI wrapper of CGI::CIPP is plugged into the Apache webserver as a Request Handler using mod_perl, which extends the

Apache webserver with a Perl interpreter. Another difference ist that the configuration options for Apache::CIPP are placed into the webserver configuration file.

All the caching is done exactly like CGI::CIPP does. See the chapter about Apache::CIPP configuration for details.

new.spirit

new.spirit uses CIPP in a different way. new.spirit is a web based development environment for creating software projects based on CIPP. In this environment the Perl code generated by CIPP for each page will be stored as a CGI executable, installed in a cgi-bin path of your webserver. This prevents you from installing your CIPP sources on the productive webserver system, only the preprocessed Perl code is installed there.

Another difference using CIPP with new.spirit is the naming convention for adressing CIPP programs. CGI::CIPP and Apache::CIPP use URL's as adresses, new.spirit expects a special dot-separated notation. See the chapter „Basic syntax rules" for details. For new.spirit CIPP configuration please refer to the new.spirit documentation.

Basic Syntax Rules

This chapter describes the CIPP syntax rules.

CIPP command structure

CIPP commands are embedded into HTML code, so the syntax is related to the HTML syntax. CIPP commands are written as tags, like HTML does. The main difference is that CIPP command tags begin with **<?** instead of **<**.

Like in HTML, there are two kinds of commands: single commands and block commands. Block commands have a start and end tag. A block command influences the HTML respectively CIPP code surrounded by it.

```
<?COMMAND [ par=value ... ] >
```

or

```
<?COMMAND [ par=value ... ]>
  HTML or CIPP code
<?/COMMAND [ par=value ... ]>
```

Whitespaces between **<?** and **COMMAND** are ignored. The command names are not case sensitive. Parameters are written als par=value pairs. Assigning a value to a parameters is optional. A parameter without a value is called a switch.

A parameter with value has the following syntax:

```
parameter_name = parameter_value
```

Whitespaces before and behind the = sign are ignored. If the value you want to assign contains whitespaces you must quote the value using double quotes.

```
<?COMMAND par_1=value_without_whitespaces
          par_2="value with whitespaces">
```

If your value contains double quotes you must escape them using the backslash character.

```
<?COMMAND par_2="value with \"double quotes\"">
```

You may place Perl variables inside your value string, they are expanded in the usual way (there is one exception regarding return parameters, see section below).

A switch without a value has this simple syntax:

```
<?COMMAND SWITCH_NAME>
```

Case sensitivity of CIPP parameters

Due to historical reasons parameter names are also not case sensitive. Actually the CIPP preprocessor converts all parameter names to lower case at a very early stage. So the exact case notation of the parameters is lost for later processing. This is usually no problem and works as you expect. HTML behaves the same.

Important Note: This approach has some side effects which you need to be aware of. For certain CIPP commands, you will be expected to specify Perl variables in the same syntactical manner of CIPP parameters. Not matter what you do, CIPP

will always work on the lower case version of these names - without giving you any warning.

The CIPP commands affected by this are: **<?MY>**, **<?INCLUDE>**, **<?GETURL>** and **<?HIDDENFIELDS>**. Please refer to the CIPP Reference chapter for details about these commands.

Important Hint: Always use lower case variable names!

CIPP return parameters

There are many CIPP commands that return parameters back to you. Since commands are inside tags, there is no way to use them in an assignment. This means that you have to specify a variable (or more than one) which should hold the return values.

These return variables are treated different from input variables.

```
$foo = "whatever"; $bar = "x";
...
<?COMMAND input=$foo output=$bar>
```

is the same as

```
<?COMMAND input="whatever" output=$bar>
```

but no the same as

```
<?COMMAND input=$foo output="x">
```

So, the return value from the command will be placed inside $bar. You cannot see from the syntax alone which parameter is expanded and which isn't. However, for each CIPP command there is a desription of return parameters (if there are any).

Context of CIPP commands

There are three different contexts which CIPP knows. They are listed and explained below. CIPP switches from one context to another only by certain block commands. Normal CIPP commands do not change the context.

1. HTML

   This is the default context your CIPP program is in. That means, if your program does not contain any CIPP commands, you will produce a simple, static HTML page.

   Inside HTML context, Perl variables are expanded with their content, like Perl does it if you use variables in a double quoted string.

   In fact HTML contexts are translated to a Perl print command, which prints the whole HTML block using some kind of double quotes.

   You can force the HTML context using the **<?HTML>** command, if you are in a Perl context (see below).

2. Variable Assignment

   This is a special context which is only existent inside of a **<?VAR>** block. Inside this block no other CIPP commands are allowed. Perl variables will be expanded. Perl expressions are also possible - see the command description for details.

   With **<?/VAR>** you terminate the assignment block and CIPP goes back to HTML context.

3. Perl

   The block command **<?PERL>** switches to this context. The whole block will be interpreted as pure Perl code. No automatic HTML output is done here, you have to use print yourself to do that. You may also use only certain CIPP commands inside a Perl block, which are **<?INCLUDE>** and **<?SQL>**. This list of such commands will be expanded in future.

   With **<?/PERL>** you terminate the command block and CIPP goes back to HTML context.

Add comments to your source

CIPP uses a similar mechanism for writing comments like Perl does. Each line which begins with a **#** sign is interpreted as a comment and is fully ignored. Leading whitespace is ignored; you're free to indent your comments.

It is not possible to preceed a CIPP comment by a CIPP command or HTML code. This would prevent you from using # in HTML code (and the least things that we want is to mess up HTML code - that is any more than it already is).

You can use the CIPP command **<?#>** for nestable multiline comments.

These lines show valid CIPP comments:

```
<?PERL>
  # this is indented comment
<?/PERL>
# this comment is not indented

<?#>
  this is a multiline comment
<?/#>
```

The following example is invalid. The comment will be printed, because it is interpreted in HTML context (see section above about HTML context).

```
<?PERL> $path = '//' <?/PERL> # setting the path
```

The corresponding web page will contain your comment:

```
# setting the path
```

Error messages

There are two kinds of error messages a CIPP developer must handle, depending on the stage the error occured: in CIPP preprocessing or Perl execution. Both stages have their own error messages.

CIPP errors

These errors occur while translating your CIPP code to Perl. They regard only the CIPP syntax, no Perl syntax checking is done at this stage. The corresponding error messages and line numbers point to the appropriate sections of your CIPP program. In CGI::CIPP and Apache::CIPP environments you'll get a HTML page with the CIPP error messages. The source code is printed out with the according sections highlighted.

Perl errors

Perl errors occur while executing the Perl program, which has been generated by CIPP. There are two classes of Perl errors: compiler and runtime errors.

Normally, a compiler error in a CGI program results in a „Server Error“, if you execute it on your webserver. The error messages may be written to the webserver error log file, depending on your webserver software and configuration.

With CIPP generated programs you should never see a „Server Error“. All CIPP environments (CGI::CIPP, Apache::CIPP and new.spirit) initiate a Perl syntax check after translating the CIPP code and before executing the Perl code the first time. Perl compiler errors are caught this way and a HTML error page is generated for you. This saves you the hassle of digging into your webserver error log file for detailed information.

Runtime errors are caught by the CIPP execption handler and can appear in different ways, depending on the location inside your program, where the error occurs. The exception handler prints out the error message, at the actual position, where the error occured. Maybe you produced already some HTML output, the error message will appear right beyond it. If you’re using some complex table layout, it can happen, that your webbrowser is unable to render the page correctly and the error message is invisible due to this. You have to look into the produced HTML source code to see the error message in this case.

All Perl error messages refer to the generated Perl code, not to your CIPP code. So line numbers are not comparable with the line numbers of your CIPP program.

CIPP preprocessor commands

There are several preprocessor commands. Those commands always begin with an exclamation mark:

```
<?!COMMAND [ par=value ... ] >
```

or

```
<?!COMMAND [ par=value ... ]>
  ...
<?/!COMMAND>
```

The special about these commands is, that they take effect at the preprocessor time and not at runtime. They modify the internal state of the preprocessor and do not create Perl code directly, like most of the other CIPP commands do.

Due to this the lexical environment of preprocessor commands does not matter the usual way. E.g. you may want to place a **<?!AUTOPRINT>** command inside of an **<?IF>** block to advice the preprocessor to generate print statements for HTML blocks or not (see the description of the **<?!AUTOPRINT>** command for details). But this will not work. See this example:

```
<?IF COND="$user_wants_an_image_file">
  <?!AUTOPRINT OFF>
  <?PERL>
    print "Content-Type: image/gif\n\n";
    system ("cat /tmp/image.gif");
  <?/PERL>
<?ELSE>
  Ok, you want no image, so you will
  get some nice <b>html</b> code.
<?/IF>
```

Looks ok but will not work!

1. The **<?AUTOPRINT>** command causes CIPP not to generate any HTTP headers for you. So the **<?ELSE>** block will not work, because no HTTP headers are printed. You'll get a 500 Server Error.

2. But even if you print headers there (with „Content-type: text/html"): the HTML block will not be printed either. The **<?!AUTOPRINT>** command does not care about the logical context. The preprocessor reads the file from the top to the bottom and will switch off autoprinting when recognizing the **<?!AUTOPRINT OFF>**. It will not be switched on and the end of the **<?IF>** block. Autoprinting will be disabled for the rest of the file. So the HTML code inside the **<?ELSE>** block will never be printed out.

So use preprocessor commands with care and keep this special implementation always in mind. Each preprocessor command description in this manual will give you hints about the corresponding special behaviour.

CHAPTER 2

CIPP Configuration

This chapter describes the configuration details for usage in connection with CGI::CIPP and Apache::CIPP. If you use CIPP in conjunction with new.spirit please refer to the according section of the new.spirit handbook.

Please check also the documentation of the corresponding Perl modules (CGI::CIPP and Apache::CIPP).

CGI::CIPP

CGI::CIPP is a Perl module which enables you to use CIPP on every CGI capable webserver. It is based on a central wrapper script, which does all the preprocessing. It executes the generated Perl code directly afterwards. Additionally, it implements a filesystem based cache for the generated code. Preprocessing is done only when the corresponding CIPP source code changed on disk, otherwise this step is skipped.

First fetch CGI::CIPP from your next CPAN mirror and install it the usual way (perl Makefile.PL; make test; make install).

Now create a CGI program in a directory, where CGI programs usually reside on your server (e.g. /cgi-bin/cipp), or configure this program another way to be a CGI program.

This program is the central CGI::CIPP wrapper. It only consists of a single function call to the CGI::CIPP module, with a hash of parameters for configuration. This is a example:

```
#!/usr/local/bin/perl

# The URL of this program is /cgi-bin/cipp

use strict;
use CGI::CIPP;

CGI::CIPP->request (
      document_root  => '/www/cippfiles',
      directoy_index => 'index.cipp',
      cache_dir      => '/tmp/cipp_cache',
      databases      => {
            test => {
                  data_source => 'dbi:mysql:test',
                  user        => 'dbuser',
                  password    => 'dbpassword',
                  auto_commit => 1
            },
            foo => {
                  ...
            }
      }
      default_database => 'test',
      lang => 'EN'
);
```

A brief description of the parameters passed to the **`CGI::CIPP->request`** call follows:

`document_root`
: This is the base directory where all your CIPP files resides. You will place CIPP programs, Includes and Config files inside this subdirectory. Using subdirectories is permitted.

  Beware that if you place your CIPP files into a subdirectory of your webservers document root, you risk that someone can fetch your CIPP source files, if he knows the URL of your CIPP document root. If you do not use the mod_rewrite configuration explained beyond, you never should place your CIPP files into your webservers document root. There is no advantage of doing this.

`directory_index`
: If you want CGI::CIPP to treat a special filename as a directory index file, pass this filename here. If you access a directory with CGI::CIPP and a according index file is found there, it will be executed.

`cache_dir`
: This names the directory where CGI::CIPP can store the preprocessed CIPP programs. If the directory does not exist it will be created. Aware, the the directory must have write access for the user under which your webserver software is running.

`databases`
: This parameter contains a hash reference, which defines several database configurations. The key of this hash is the CIPP internal name of the database, which can be addressed by the **`DB`** parameter of all CIPP SQL commands. The value is a hash reference with the following keys defined.

  `data_source`
  : This must be a DBI conforming data source string. Please refer to the DBI documentation for details about this.

  `user`
  : This is the username CIPP uses to connect to the database

  `password`
  : This password is used for the database user.

auto_commit	This parameter sets the initial state of the Auto-Commit flag. Please refer to the description of the **<?AUTOCOMMIT>** command or the DBI documentation for details about AutoCommit.
default_database	This takes the name of the default database. This database is always used, if a CIPP SQL command ommits the **DB** parameter. The value passed here must be a defined key in the **databases** hash.
lang	CIPP has multilanguage support for its error messages, actually english (**'EN'**) and german (**'DE'**) are supported.

The CGI wrapper program uses the CGI feature **PATH_INFO** to determine which page should be executed. To execute the CIPP page 'test.cipp' located in '/www/htdocs/cippfiles/foo/test.cipp' you must specify the following URL (assuming the configuration of the example above):

```
http://somehost/cgi-bin/cipp/foo/test.cipp
```

You simply add the path of your page (relative to the path you specified with the **document_root** parameter) to the URL of the CGI wrapper.

Be aware of the real URL of your page if you use relative URL's to non CIPP pages. In the above example relative URL's must consider that the CGI wrapper program is located in a different location as the directory you declared as the CIPP document root. To avoid confusion about this, you should configure your webserver in that way, that the CGI wrapper program has a URL which is located inside your webservers document root. This way using relative URLs is easier, because you never left the document root of your webserver.

If you're using the Apache webserver (what is always recommended :) you have several alternatives of doing this.

- using a extra ScriptAlias
- using mod_rewrite

Using a extra ScriptAlias

This is a example configuration of using a ScriptAlias to configure CIPP for easy usage of relative URLs.

These are the corresponding basic Apache configuration parameters:

```
DocumentRoot      "/www/htdocs"
ScriptAlias       "/cipp" "/www/cgi-bin/cipp"
```

Now the CGI wrapper program URL is located inside your document root. This is a example URL for a CIPP page located in /www/htdocs/cipp/foo/test.cipp

```
http://somehost/cipp/foo/test.cipp
```

The disadvantage of this configuration is, that your CIPP root directory /www/cipp-files cannot contain other files than CIPP files. It is not possible to put images or static HTML documents here, because you cannot reach these documents with normal URLs.

Using mod_rewrite

You avoid the above mentioned disadvantage if you use mod_rewrite. These are the corresponding basic Apache configuration parameters (please refer to the Apache documentation for details). You will need Apache version 1.2.x or better for using mod_rewrite.

```
DocumentRoot      "/www/htdocs"
ScriptAlias       "/cgi-bin" "/www/cgi-bin"
RewriteEngine     "on"
RewriteRule       "^/(.*\.cipp.*)"
                  "/cgi-bin/cipp/$1" [PT]
```

The CGI wrapper program is still located in a extra cgi-bin directory. But the **RewriteRule** directs all URL's with the suffix .cipp, no matter where they are located, to the CIPP CGI wrapper program.

Now we have to change the CGI::CIPP configuration:

```
document_root     /www/htdocs
```

This is slightly different. We now declare the Apache **DocumentRoot** also to be the **document_root** of CGI::CIPP, so no special subdirectory is needed. The Apache rewrite engine is responsible for translating URL's with the suffix .cipp to a appropriate call of the CGI wrapper program.

This is a example URL for a CIPP page located in /www/htdocs/foo/test.cipp

```
http://somehost/foo/test.cipp
```

Now you are able to place CIPP files on your webserver wherever you want, because there is no special CIPP directory anymore. Only the suffix .cipp is relevant, due to the RewriteRule above.

CGI::SpeedyCGI and CIPP::CGI

There exists a really nice module called CGI::SpeedyCGI, which is available freely via CPAN. It implements a nifty way of making Perl CGI processes persistent, so subseqeuent CGI calls are answered much more faster.

Using CIPP::CGI together with CGI::SpeedyCGI is easy. Simply replace the perl interpreter path in the shebang line **`#!/usr/local/bin/perl`** with the according path to the speedy program, e.g.: **`#!/usr/local/bin/speedy`**.

Refer to the CGI::SpeedyCGI documentation for details about configuring SpeedyCGI. We recommend the usage of the -r and -t switch, so you are able to control the number of parallel living speedy processes, e.g.

```
#!/usr/local/bin/speedy -- -r30 -t120
```

Each speedy process now answeres a maximum of 30 requests and then dies. If a process is idle for longer than 120 secs it dies also.

Apache::CIPP

If you use the Apache::CIPP module you have all advantages of a CGI::CIPP / CGI::SpeedyCGI configuration, particular regarding the persistence stuff. Also configuration of Apache::CIPP is simplier, because you put all required parameters into the Apache configuration file(s). All above mentioned CGI::CIPP configuration tasks to make CGI::CIPP work as transparently as possible are needless in a Apache::CIPP environment.

Apache::CIPP needs the Apache module mod_perl to run, so first fetch mod_perl and Apache::CIPP from your next CPAN mirror and install them.

This is a example of the section you have to add to your httpd.conf:

```
<Location ~ ".*\.cipp">
SetHandler "perl-script"
PerlHandler Apache::CIPP

PerlSetVar cache_dir      /tmp/cipp_cache
PerlSetVar debug          1
PerlSetVar lang           DE
PerlSetVar databases      test, foo
PerlSetVar default_db     test

PerlSetVar db_test_data_source  dbi:mysql:test
PerlSetVar db_test_user         dbuser
PerlSetVar db_test_password     dbpassword
PerlSetVar db_test_auto_commit  1

PerlSetVar db_foo_data_source   dbi:Oracle:foo
PerlSetVar db_test_user         dbuser2
PerlSetVar db_test_password     dbpassword2
PerlSetVar db_test_auto_commit  1
</Location>
```

The regular expression inside the **<Location>** tag matches all files with the suffix .cipp, independent from the location on your server. Due to this you are able to place your CIPP pages everywhere you want. So mixing of .cipp and other files is no problem with this configuration.

Apache::CIPP Parameters are very similar to the CGI::CIPP parameters:

cache_dir This names the directory where CGI::CIPP can store the preprocessed CIPP programs. If the directory does not exist it will be created. Aware, the the directory must have write access for the user under which your webserver software is running.

debug If you set the **debug** parameter to non zero, each request to Apache::CIPP will be logged in the apache error logfile, together with some information about the internal state of the caches.

databases This parameter lists the CIPP internal names of all your database configurations. The list is comma separated, whitspace is ignored.

default_database This takes the name of the default database. This database is always used, if a CIPP SQL command ommits the **DB** parameter. The value passed here must be a defined value in the **databases** parameter.

lang CIPP has multilanguage support for its error messages, actually english (**'EN'**) and german (**'DE'**) are supported.

The following four parameters must be specified for each database you listed in the **databases** parameter. Replace the ***** with the appropriate database name.

db_*_data_source This must be a DBI conforming data source string. Please refer to the DBI documentation for details about this.

db_*_username This is the username CIPP uses to connect to the database

db_*_password This password is used for the database user.

db_*_auto_commit This parameter sets the initial state of the AutoCommit flag. Please refer to the description of the **<?AUTOCOMMIT>** command or the DBI documentation for details about AutoCommit.

CHAPTER 3

CIPP Command Reference

This chapter describes all CIPP commands in alphabetical order. Each reference contains syntax notation, textual description and examples for each command.

Command Groups

For better overview the following table lists all CIPP commands grouped by type:

Variables and Scoping

VAR	Definition of a variable
MY	Declaring a private (block local) variable
BLOCK	Creation of a block context to limit the scope of private variables

Control Structures

IF	Conditional execution of a block
ELSIF	Subsequent conditional execution
ELSE	Alternative execution of a block
WHILE	Loop with condition check before first iteration
DO	Loop with condition check after first iteration
FOREACH	Loop iterating with a variable over a list
PERL	Insertion of pure Perl code
SUB	Definition of a Perl subroutine

Import

INCLUDE	Insertion of a CIPP Include file in the actual CIPP code
MODULE	Definition of a CIPP Perl Module
REQUIRE	Import a CIPP Perl Module
USE	Import a standard Perl module
CONFIG	Import a config file

Exception Handling

TRY	Secured execution of a block. Any exceptions thrown in the encapsulated block are caught.
CATCH	Execution of a block if a particular exception was thrown in a preceding TRY block.
THROW	Explicite creation of an exception.
LOG	Write a entry in a logfile.

SQL

SQL	Execution of a SQL statement
COMMIT	Commit a transaction
ROLLBACK	Rollback a transaction
AUTOCOMMIT	Control of transaction behaviour
DBQUOTE	Quoting of a variable for usage in a SQL statement
GETDBHANDLE	Returns the internal DBI database handle

URL- and Form Handling

GETURL	Creation of a CIPP object URL
URLENCODE	URL encoding of a variable
HTMLQUOTE	HTML encoding of a variable
HIDDENFIELDS	Producing a number of hidden formular fields

HTML Tag Replacements

A	Replaces <A> tag
FORM	Replaces <FORM> tag
IMG	Replaces <IMG> tag
OPTION	Replaces <OPTION> tag, with sticky feature
INPUT	Replaces <INPUT> tag, with sticky feature
TEXTAREA	Replaces <TEXTAREA> tag
SELECT	Replaces <SELECT> Tag, with sticky feature

Interface

INTERFACE	Declaration of a CGI interface for a CIPP program
INCINTERFACE	Declaration of a interface for CIPP Include
GETPARAM	Recieving a non declared CGI input parameter
GETPARAMLIST	Returns a list of all CGI input parameter names
SAVEFILE	Storing a client side upload file

Apache

APGETREQUEST	Returns the internal Apache request object
APREDIRECT	Redirects to another URL internally

Preprocessor

!AUTOPRINT	Controls automatic output of HTML code
!HTTPHEADER	Dynamic generation of a HTTP header
!PROFILE	Initiate generation of profiling code

Debugging

DUMP	Dumps preformatted contents of data structures

Alphabetical Reference

#

Type

Multi Line Comment

Syntax

```
<?#>
...
<?/#>
```

Description

This block command realizes a multiline comment. Simple comments are introduced with a single # sign, so you can only comment one line with them. All text inside a <?#> block will be treated as a comment and will be ignored. Nesting of <?#> is allowed.

Example

This is a simple multi line comment.

```
<?#>
  This text will be ignored.
  All CIPP tags too.
  So this is no syntax error
  <?IF foo bar>
<?/#>
```

You may nest <?#> blocks:

```
<?#>
  bla foo
  <?#>
     foo bar
  <?/#>
<?/#>
```

A

Type

HTML Tag Replacement

Syntax

```
<?A HREF=hyperlinked_object_name[#anchor]
    [ additional_<A>_parameters ... ] >
...
<?/A>
```

Description

This command replaces the <A> HTML tag. You will need this in a new.spirit environment to set a link to a CIPP CGI or HTML object.

Parameter

HREF

This parameter takes the name of the hyperlinked object. You may optionally add an anchor (which should be defined using <**A NAME**> in the referred page) using the **#** character as a delimiter.

This paremeter is expected as an URL in CGI::CIPP or Apache::CIPP environments and in dot-separated object notation in a new.spirit environment.

additional_<A>_parameters

All additional parameters are taken into the generated <**A**> tag.

Example

Textual link to 'MSG.Main', in a new.spirit environment.

```
<?A HREF="MSG.Main">Back to the main menu<?/A>
```

Image link to '/main/menu.cgi', in a CGI::CIPP or Apache::CIPP environment:

```
<?A HREF="/main/menu.cgi">
<?IMG SRC="/images/logo.gif" BORDER=0>
<?/A>
```

APGETREQUEST

Type

Apache

Syntax

<?**APGETREQUEST** [**MY**] **VAR**=*request_variable* >

Description

This command is only working if CIPP is used as an Apache module.

It returns the internal Apache request object, so you can use Apache specific features.

Parameter

VAR
This is the variable where the request object will be stored.

MY
If you set the **MY** switch, the created variable will be declared using 'my'. Its scope reaches to the end of the block which surrounds the **APGETREQUEST** command.

Example

The Apache request object will be stored in the implicitely declared variable $ar.

```
<?APGETREQUEST MY VAR=$ar>
```

APREDIRECT

Type

Apache

Syntax

<?**APREDIRECT URL**=*new_URL* >

Description

This command is only working if CIPP is used as an Apache module.

It results in an internal Apache redirect. That means, the new url will be 'executed' without notifying the client about this.

Parameter

URL
This expression is used for the new URL.

Note:

The program which uses **<?APREDIRECT>** should not produce any output, otherwise this may confuse the webserver or the client, if more then one HTTP header is sent. So you should use **<?AUTOPRINT OFF>** at the top of the program to circumvent that.

Example

This commands redirect internally to the homepage of the corresponding website:

```
<?AUTOPRINT OFF>
<?APREDIRECT URL="/">
```

AUTOCOMMIT

Type

SQL

Syntax

```
<?AUTOCOMMIT ( ON | OFF )
             [ DB=database_name ]
             [ DBH=database_handle ]
             [ THROW=exception ] >
```

Description

The **<?AUTOCOMMIT>** command corresponds directly to the underlying DBI AutoCommit mechanism.

If AutoCommit is activated each SQL statement will implicitely be executed in its own transaction. Think of a **<?COMMT>** after each statement. Explicite use of **<?COMMIT>** or **<?ROLLBACK>** is forbidden in AutoCommit mode.

If AutoCommit is deactivated you have to call **<?COMMIT>** or **<?ROLL-BACK>** yourself. CIPP will rollback any uncommited open transactions at the end of the program.

Parameter

ON|OFF
Switch AutoCommit modus either on or off.

DB
This is the CIPP internal name of the database for this command. In CGI::CIPP or Apache::CIPP environment this name has to be defined in the appropriate global configuration. In a new.spirit environment this is the name of the database configuration object in dot-separated notation.

If DB is ommited the project default database is used.

DBH
Use this option to pass an existing DBI database handle, which should used for this SQL command. You can't use the **DBH** option in conjunction with **DB**.

THROW

With this parameter you can provide a user defined exception which should be thrown on failure. The default exception thrown by this statement is **autocommit**.

If the underlying database is not capable of transactions (e.g. MySQL) setting AutoCommit to **ON** will throw an exception.

Example

Switch AutoCommit on for the database 'foo'.

```
<?AUTOCOMMIT ON DB="foo">
```

Switch AutoCommit off for the database 'bar' and throw the user defined exception 'myautocommit' on failure.

```
<?AUTOCOMMIT OFF DB="bar" THROW="myautocommit">
```

!AUTOPRINT

Type

Preprocessor

Syntax

<?!**AUTOPRINT OFF**>

Description

With the **<?!AUTOPRINT OFF>** command the preprocessor can be advised to suppress the generation of print statements for non CIPP blocks. The default setting is **ON** and it is only possible to switch it **OFF** and not the other way around.

In earlier versions of CIPP this command was named **<?AUTOPRINT>**. This notation is depreciated, but will work for compatability reasons.

Parameter

OFF
Automatic generation of print statements for non CIPP blocks will be deactivated.

Note

This is a preprocessor command. Please read the chapter about preprocessor commands for details about this.

You should use this command at the very top of your program file. CIPP will not generate any HTTP headers for you, if you use **<?!AUTOPRINT OFF>**, so you have to do this on your own. If you only want to generate a special HTTP header, use **<?!HTTPHEADER>** instead.

The „CIPP Introduction“ Chapter contains a paragraph about CIPP Preprocessor Commands. Please refer to this discussion for details of **<?!AUTOPRINT>**.

Example

This program sends a GIF image to the client, after generating the proper HTTP header. (For another example, see **<?APREDIRECT>**)

```
<?AUTOPRINT OFF>
These lines will never be printed, they are fully
ignored!!!
<?PERL>
  my $file = “/tmp/image.gif“;
  my $size = -s $file;

  print “Content-type: image/gif\n“;
  print “Content-length: $size\n\n“;

  open (GIF, $file) or die “can’t open $file“;
  while (<GIF>) {
    print;
  }
  close GIF;
<?/PERL>
```

BLOCK

Type

Variables and Scoping

Syntax

```
<?BLOCK>
...
<?/BLOCK>
```

Description

Use the **<?BLOCK>** command to divide your program into logical blocks to control variable scoping. Variables declared with **<?MY>** inside a block are not valid outside.

Example

The variable $example does not exist beyond the block.

```
<?BLOCK>
  <?MY $example>
  $example is known.
<?/BLOCK>

$example does not exist here. This will
result in a Perl compiler error, because
$example is not declared here.
```

CATCH

Type

Exception Handling

Syntax

```
<?CATCH [ THROW=exception ]
        [ MY ]
        [ EXCVAR=variable_for_exception ]
        [ MSGVAR=variable_for_error_message ] >
...
<?/CATCH>
```

Description

Typically a **<?CATCH>** block follows after a **<?TRY>** block. You can process one particular or just any exception with the **<?CATCH>** block.

<?CATCH> and **<?TRY>** has to be placed inside the same block.

See the description of **<?TRY>** for details about the CIPP exception handling mechanism.

Parameter

THROW
If this parameter is omitted, all exceptions will be processed here. Otherwise the **<?CATCH>** block is executed only if the appropriate exception was thrown.

EXCVAR
Names the variable, where the exception identifier should be stored in. Usefull if you use **<?CATCH>** for a generic exception handler and omitted the **THROW** parameter.

MSGVAR
Name the variable, where the error message should be stored in.

MY
If you set the **MY** switch the created variable will be declared using 'my'. Its scope reaches to the end of the block which surrounds the **<?CATCH>** command.

Example

We try to insert a row into a database table, which has a primary key defined, and commit the transcation. We catch two exceptions: the possible primary key constraint violation and a possible commit exception, maybe the database is not capable of transactions.

```
<?TRY>
  <?SQL SQL="insert into persons
            (firstname, lastname)
             values ('John', 'Doe')"><?/SQL>
  <?COMMIT>
<?/TRY>

<?CATCH THROW=sql MY MSGVAR=$message>
  <?LOG MSG="Can't insert data: $message"
        TYPE="database">
<?/CATCH>

<?CATCH THROW=commit MSGVAR=$message>
  <?LOG MSG="COMMIT rejected: $message"
        TYPE="database">
<?/CATCH>
```

COMMIT

Type

SQL

Syntax

```
<?COMMIT [ DB=database_name ]
         [ DBH=database_handle ]
         [ THROW=exception ] >
```

Description

The **<?COMMIT>** command concludes the actual transaction and makes all changes to the database permanent.

Using **<?COMMIT>** in **<?AUTOCOMMIT ON>** mode is not possible.

If you are not in **<?AUTOCOMMIT ON>** mode a transaction begins with the first SQL statement and end either with a **<?COMMIT>** or **<?ROLLBACK>** command.

Parameter

DB

This is the CIPP internal name of the database for this command. In CGI::CIPP or Apache::CIPP environment this name has to be defined in the appropriate global configuration. In a new.spirit environment this is the name of the database configuration object in dot-separated notation.

If DB is ommited the project default database is used.

DBH

Use this option to pass an existing DBI database handle, which should used for this SQL command. You can't use the **DBH** option in conjunction with **DB**.

THROW

With this parameter you can provide a user defined exception which should be thrown on failure. The default exception thrown by this statement is **commit**.

If the underlying database is not capable of transactions (e.g. MySQL) execution of this command will throw an exception.

Example

We insert a row into a database table and commit the change immediately. We throw a user defined exeption, if the commit fails. So be safe we first disable AutoCommiting.

```
<?AUTOCOMMIT OFF>
<?SQL SQL="insert into foo (num, str)
           values (42, 'bar');">
<?/SQL>
<?COMMIT THROW="COMMIT_Exception">
```

CONFIG

Type

Import

Syntax

```
<?CONFIG NAME=config_file
         [ RUNTIME ] [ NOCACHE ]
         [ THROW=exception ] >
```

Description

The **<?CONFIG>** command reads a config file. This is done via a mechanism similar to Perl's require, so the config file has to be pure Perl code defining global variables.

<?CONFIG> ensures a proper load of the configuration file even in persistent Perl environments.

In contrast to "require" **<?CONFIG>** will reload a config file when the file was altered on disk. Otherwise the file will only be loaded once.

Parameter

NAME

This is the name of the config file, expected as an URL in CGI::CIPP or Apache::CIPP environments and in dot-separated object notation in a new.spirit environment.

RUNTIME

This switch makes sense only in a new.spirit environment. If you set it the **NAME** parameter will be resolved at runtime, so it can contain variables. new.spirit will not check the existance of the file in this case. Normally you'll get a CIPP error message, if the adressed file does not exist.

In CGI::CIPP and Apache::CIPP environments the **NAME** parameter will always be resolved at runtime.

NOCACHE

This switch is useful in persistant Perl environments. It forces **<?CONFIG>** to read the config file even if it did not change on disk. You'll need this if your config file does some calculations based on the request environment, e.g. if the value of some variables depends on the clients user agent.

THROW

With this parameter you can provide a user defined exception to be thrown on failure. The default exception thrown by this statement is **config**.

An exception will be thrown, if the config file does not exist or is not readable.

Example

Load of the configuration file "/lib/general.conf", with disabled cache, used in CGI::CIPP or Apache::CIPP environment:

```
<?CONFIG NAME="/lib/general.conf" NOCACHE>
```

Load of the configuration file object x.custom.general in a new.spirit environment:

```
<?CONFIG NAME="x.custom.general">
```

Load of a config file with a name determined at runtime, in a new.spirit environment, throwing "myconfig" on failure:

```
<?CONFIG NAME="$config_file" RUNTIME
         THROW="myconfig">
```

DBQUOTE

Type

SQL

Syntax

```
<?DBQUOTE VAR=variable
          [ MY ]
          [ DBVAR=quoted_result_variable ]
          [ DB=database_name ]
          [ DBH=database_handle ] >
```

Description

<?SQL> (and DBI) has a nice way of quoting parameters to SQL statements (called parameter binding). Usage of that mechanism is generally recommended (see **<?SQL>** for details). However if you need to construct your own SQL statement, **<?DBQUOTE>** will let you do so.

<?DBQUOTE> will generate the string representation of the given scalar variable as fit for an SQL statement. That is, it takes care of quoting special characteres.

Parameter

VAR
This is the scalar variable containing the parameter you want to be quoted.

DBVAR
This optional parameters takes the variable where the quoted content should be stored. The surrounding ' characters are part of the result, if the variable is not undef. A value of undef will result in **NULL** (without the surrounding '), so the quoted variable can be placed directly in a SQL statement.

If you ommit **DBVAR**, the name of the target variable is computed by placing the prefix 'db_' in front of the **VAR** name.

MY
If you set the **MY** switch the created variable will be declared using 'my'. Its scope reaches to the end of the block which surrounds the **<?DBQUOTE>** command.

DB

This is the CIPP internal name of the database for this command. In CGI::CIPP or Apache::CIPP environment this name has to be defined in the appropriate global configuration. In a new.spirit environment this is the name of the database configuration object in dot-separated notation.

If DB is ommited the project default database is used.

DBH

Use this option to pass an existing DBI database handle, which should used for this SQL command. You can't use the **DBH** option in conjunction with **DB**.

Example

This quotes the variable $name, the result will be stored in the just declared variable $db_name.

```
<?DBQUOTE MY VAR="$name">
```

This quotes $name, but stores the result in the variable $quoted_name.

```
<?DBQUOTE VAR="$name" MY DBVAR="$quoted_name">
```

The quoted variable can be used in a SQL statement this way:

```
<?SQL SQL="insert into persons (name)
           values ( $quoted_name )">
```

DO

Type

Control Structure

Syntax

```
<?DO>
...
<?/DO COND=condition >
```

Description

The **<?DO>** block repeats executing the contained code as long as the condition evaluates true. The condition is checked afterwards. That means that the block will always be executed at least once.

Parameter

COND

This takes a Perl condition. As long as this condition is true the **<?DO>** block will be repeated.

Example

Print "Hello World" $n times. (note: for n=0 and n=1 you get the same result)

```
<?DO>
  Hello World<BR>
<?/DO COND="--$n > 0">
```

DUMP

Type

Debugging

Syntax

```
<?DUMP $var_1 ... $var_n>
```

Description

The **<?DUMP>** command dumps the contents of the given variables using Data::Dumper, inside of a HTML <pre></pre> block.

Parameter

$var_1 .. $var_n
The contents of this variables are dumped to STDOUT.

Example

```
<?DUMP $hash_ref $list_ref>
```

ELSE

Type

Control Structure

Syntax

<?**ELSE**>

Description

<?ELSE> closes an open **<?IF>** or **<?ELSIF>** conditional block and opens a new block (which is later terminated by **<?/IF>)**. The block is only executed if the condition of the preceding block was evaluated and failed.

<?MY> variables are only visible inside this block.

(Or short: it works as you would expect.)

Example

Only Larry gets a personal greeting message:

```
<?IF COND="$name eq 'Larry'">
  Hi Larry, you're welcome!
<?ELSE>
  Hi Stranger!
<?/IF>
```

ELSIF

Type

Control Structure

Syntax

<?**ELSIF COND**=*condition* >

Description

<?ELSIF> closes an open **<?IF>** or **<?ELSIF>** conditional block and opens a new block. The condition is only evaluated if the condition of the preceding block was evaluated and failed.

<?MY> variables are only visible inside this block.

(Or short: it works as you would expect.)

Parameter

COND
Takes the Perl condition.

Example

Larry and Linus get personal greeting messages:

```
<?IF COND="$name eq 'Larry'">
  Hi Larry, you're welcome!
<?ELSIF COND="$name eq 'Linus'">
  Hi Linus, you're velkomma!
<?ELSE>
  Hi Stranger!
<?/IF>
```

FOREACH

Type

Control Structure

Syntax

```
<?FOREACH [ MY ] VAR=running_variable
          LIST=perl_list >
...
<?/FOREACH>
```

Description

<?FOREACH> corresponds directly the Perl foreach command. The running variable will iterate of the list, executing the enclosed block for each value of the list.

Parameter

VAR

This is the scalar running variable.

LIST

You can write any Perl list here, e.g. using the bracket notation or pass a array variable using the @ notation.

MY

If you set the **MY** switch the created running variable will be declared using 'my'. Its scope reaches to the end of the block which surrounds the **<?FOREACH>** command.

Note: this is a slightly different behaviour compared to a Perl "foreach my $var (@list)" command, where the running variable $var is valid only inside of the foreach block.

Example

Counting up to ’three’:

```
<?FOREACH MY VAR=“$cnt“
          LIST=“(’one’, ’two’, ’three’)“>
  $cnt
<?/FOREACH>
```

FORM

Type

HTML Tag Replacement

Syntax

```
<?FORM ACTION=cgi_file
       [ additional_<FORM>_parameters ... ] >
...
<?/FORM>
```

Description

<?FORM> generates a HTML **<FORM>** tag, setting the **ACTION** option to the appropriate URL. The request METHOD defaults to POST if no other value is given.

Parameter

ACTION
This is the name of the form target CGI program, expected as an URL in CGI::CIPP or Apache::CIPP environments and in dot-separated object notation in a new.spirit environment.

additional_<FORM>_parameters
All additional parameters are taken over without changes into the produced <**FORM**> tag. If you ommit the **METHOD** parameter it will default to POST.

Example

Creating a named form with a submit button, pointing to the CGI object “x.login.start“, in a new.spirit environment:

```
<?FORM ACTION=“x.login.start“ NAME=“myform“>
<?INPUT TYPE=SUBMIT VALUE=“ Start “>
<?/FORM>
```

Creating a similar form, but the action is written as an URL because we are in CGI::CIPP or Apache::CIPP environment:

```
<?FORM ACTION="/login/start.cgi" NAME="myform">
<?INPUT TYPE=SUBMIT VALUE=" Start ">
<?/FORM>
```

GETDBHANDLE

Type

SQL

Syntax

```
<?GETDBHANDLE [ DB=database_name ] [ MY ]
              VAR=handle_variable >
```

Description

This command returns a reference to the internal Perl database handle, which is the object references returned by **DBI->connect**.

With this handle you are able to perform DBI specific functions which are currently not directly available through CIPP.

Parameter

VAR
This is the variable where the database handle will be stored.

MY
If you set the **MY** switch the created variable will be declared using 'my'. Its scope reaches to the end of the block which surrounds the **<?GETDBHANDLE>** command.

DB
This is the CIPP internal name of the database for this command. In CGI::CIPP or Apache::CIPP environment this name has to be defined in the appropriate global configuration. In a new.spirit environment this is the name of the database configuration object in dot-separated notation.

If DB is ommited the project default database is used.

Example

We get the database handle for the database object 'x.Oracle' in a new.spirit environment and perform a select query using this handle.

Ok, you simply can do this with the **<?SQL>** command, but now you can see how much work is done for you through CIPP :)

```
<?GETDBHANDLE DB="MSG.Oracle" MY VAR="$dbh">

<?PERL>
  my $sth = $dbh->prepare ( qq{
      select n,s from TEST_table
      where n between 10 and 20
  });
  die "my_sql\t$DBI::errstr" if $DBI::errstr;

  $sth->execute;
  die "my_sql\t$DBI::errstr" if $DBI::errstr;

  my ($n, $s);
  while ( ($n, $s) = $sth->fetchrow ) {
    print "n=$n s=$s<BR>\n";
  }
  $sth->finish;
  die "my_sql\t$DBI::errstr" if $DBI::errstr;

<?/PERL>
```

GETPARAM

Type

Interfaces

Syntax

```
<?GETPARAM NAME=parameter_name
           [ MY ] [ VAR=content_variable ] >
```

Description

With this command you can explicitely get a CGI parameter. This is useful if your CGI program uses dynamically generated parameter names, so you are not able to use **<?INTERFACE>** for them.

Refer to **<?INTERFACE>** to see how easy it is to handle standard CGI input parameters.

Parameter

NAME
Identifier of the CGI input parameter

VAR
This is the variable where the content of the CGI parameter will be stored. This can be either a scalar variable (indicated through a $ sign) or an array variable (indicated through a @ sign).

MY
If you set the **MY** switch the created variable will be declared using 'my'. Its scope reaches to the end of the block which surrounds the **<?GETPARAM>** command.

Example

We recieve two parameters, one staticly named parameter and one scalar parameter, which has a dynamic generated identifier.

```
<?GETPARAM NAME="listparam" MY VAR="@list">
<?GETPARAM NAME="scalar$name" MY VAR="$scalar">
```

GETPARAMLIST

Type

Interfaces

Syntax

<?**GETPARAMLIST** [**MY**] **VAR**=*variable* >

Description

This command returns a list containing the identifiers of all CGI input parameters.

Parameter

VAR
This is the variable where the identifiers of all CGI input parameters will be stored in. It must be an array variable, indicated through a @ sign.

MY
If you set the **MY** switch the created list variable will be declared using 'my'. Its scope reaches to the end of the block which surrounds the **<?GETPARAMLIST>** command.

Example

The list of all CGI input parameter identifiers will be stored into the array variable @input_param_names.

```
<?GETPARAMLIST MY VAR="@input_param_names">
```

GETURL

Type

URL and Form Handling

Syntax

```
<?GETURL NAME=object_file
         [ MY ] VAR=target_variable
         [ RUNTIME ] [ THROW=exception ] >
         [ PARAMS=parameters_variables ]
         [ PAR_1=value_1 ... PAR_n=value_n ] >
```

Description

This command returns a URL, optionally with parameters. In a new.spirit environment you use this to resolve the dot-separated object name to a real life URL.

In CGI::CIPP and Apache::CIPP environments this is not necessary, because you work always with real URLs. Nevertheless it also useful there, because its powerfull possibilities of generating parmeterized URLs.

Parameter

NAME

This is the name of the specific file, expected as an URL in CGI::CIPP or Apache::CIPP environments and in dot-separated object notation in a new.spirit environment.

VAR

This is the scalar variable where the generated URL will be stored in. In earlier versions of CIPP this option was named **URLVAR**. The usage of the URLVAR notation is depreciated, but it works for compatibility reasons. To prevent from logical errors CIPP throws an error if you use URLVAR and VAR inside of one command (e.g. to create an URL which contains a parameter called VAR or URLVAR).

URLVAR

Depreciated. See description of VAR.

MY

If you set the **MY** switch the created variable will be declared using 'my'. Its scope reaches to the end of the block which surrounds the **<?GETURL>** command.

RUNTIME

This switch makes only sense in a new.spirit environment. The **NAME** parameter will be resolved at runtime, so it can contain variables. CIPP will not check the existance of the file in this case. Normally you get a CIPP error message, if the adressed file does not exist.

In CGI::CIPP and Apache::CIPP environments the **NAME** parameter will always be resolved at runtime.

THROW

With this parameter you can define the exception to be thrown on failure. The default exception thrown by this statement is **geturl**.

An exception will be thrown, if the adressed file does not exist.

PARAMS

This takes a comma separated list of parameters, which will be encoded and added to the generated URL. You may pass scalar variables (indicated through the $ sign) and also array variables (indicated through the @ sign).

With the **PARAMS** option you can only pass parameters whose values are stored in variables with the same name (where case is significant). The variables listed in **PARAMS** will be treated case sensitive.

PAR_1..PAR_n

Any additional parameters to **<?GETURL>** are interpreted as named parameters for the URL. You can pass scalar and array values this way (using $ and @). Variables passed this way are seen by the called program as lower case written variable names, no matter which case you used in **<?GETURL>**.

Note

It is highly recommended to use lower case variable names. Due to historical reasons CIPP converts parameter names to lower case without telling you about it. If this ever gets "fixed" and you have uppercase latters, your code will break. So, use lowercase.

Example

We are in a new.spirit environment and produce a <IMG> tag, pointing to a new.spirit object (btw: the easiest way of doing this is the **<?IMG>** command):

```
<?GETURL NAME="x.Images.Logo" MY VAR=$url>
<IMG SRC="$url">
```

Now we link the CGI script "/secure/messager.cgi" in a CGI::CIPP or Apache::CIPP environment. We pass some parameters to this script. (Note the case sensitivity of the parameter names, we really should use lower case variables all the time!)

```
<?VAR MY NAME=$Username>hans<?/VAR>
<?VAR MY NAME=@id>(1,42,5)<?/VAR>
<?GETURL NAME="/secure/messager.cgi" MY VAR=$url
         PARAMS="$Username, @id" EVENT=delete>
<A HREF="$url">delete messagse</A>
```

The CGI program "/secure/messager.cgi" recieves the parameters this way (note that the **$Username** parameter is seen as **$Username**, but **EVENT** is seen as **$event**). If you find this confusing, use always lower case variable names.

```
<?INTERFACE INPUT="$event, $Username, @id">
<?IF COND="$event eq 'delete'">
  <?MY $id_text>
  <?PERL>$id_text = join (", " @id)<?PERL>
  You are about to delete
  $username's ID's?: $id_text<BR>
<?/IF>
```

HIDDENFIELDS

Type

URL and Form Handling

Syntax

```
<?HIDDENFIELDS [ PARAMS=parameter_variables ]
               [ PAR_1=value_1 ... PAR_n=value_n ] >
```

Description

This command produces a number of <INPUT TYPE=HIDDEN> HTML tags, one for each parameter you specify. Use this to transport a bunch of parameters via a HTML form. This command takes care of special characters in the parameter values and quotes them if necessary.

Parameter

PARAMS

This takes a comma separated list of parameters, which will be encoded and transformed to a <INPUT TYPE=HIDDEN> HTML tag. You may pass scalar variables (indicated through the $ sign) and also array variables (indicated through the @ sign).

With the **PARAMS** option you can only pass parameters whose values are stored in variables with the same name (where case is significant).

PAR_1..PAR_n

Any additional parameters to **<?HIDDENFIELDS>** are interpreted as named parameters. You can pass scalar and array values this way (using $ and @). Variables passed this way are seen by the called program as lower case written variable names, no matter which case you used in **<?HIDDENFIELDS>**.

Example

This is a form in a new.spirit environment, pointing to the object “x.secure.messager“. The two parameters $username and $password are passed via PARAMS, the parameter “event“ is set to “show“.

```
<?FORM ACTION=“x.secure.messager“>
<?HIDDENFIELDS PARAMS=“$username, $password“
               event=“show“>
<INPUT TYPE=SUBMIT VALUE=“show messages“>
<?/FORM>
```

HTMLQUOTE

Type

URL and Form Handling

Syntax

```
<?HTMLQUOTE VAR=variable_to_encode
            [ MY ] HTMLVAR=target_variable >
```

Description

This command quotes the content of a variable, so that it can be used inside a HTML option or <TEXTAREA> block without the danger of syntax clashes. The following conversions are done in this order:

```
&  =>  &amp;
<  =>  &lt;
“  =>  &quot;
```

Parameter

VAR
This is the scalar variable containing the parameter you want to be quoted.

HTMLVAR
This non-optional parameter takes the variable where the quoted content will be stored.

MY
If you set the **MY** switch the created variable will be declared using ’my’. Its scope reaches to the end of the block which surrounds the **<?HTMLQUOTE>** command.

Example

We produce a <TEXTAREA> tag with a quoted instance of the variable $text. Note: you can also use the **<?TEXTAREA>** command for this purpose.

```
<?HTMLQUOTE VAR=“$text“ MY HTMLVAR=“$html_text“>
<TEXTAREA NAME=“text“>$html_text</TEXTAREA>
```

<?!HTTPHEADER

Type

Preprocessor

Syntax

```
<?!HTTPHEADER [ MY ] VAR=http_header_hash_ref >
  # Perl Code which modifies the
  # http_header_hash_ref variable
<?/!HTTPHEADER>
```

Description

Use this command, if you want to modify the standard HTTP header response. CIPP generates by default a simple HTTP header of this form:

```
Content-type: text/html\n\n
```

In a new.spirit environment you can define a project wide default HTTP header extension, e.g. „Pragme: no-cache“, or something similar.

If you want to modify the HTTP header at runtime, you can use this command. The **<?!HTTPHEADER>** command switches to Perl context, so you write Perl code inside the block. The variable you declared with the VAR option is accessable inside this block and will contain a reference to a hash containing the default HTTP header tags. Your Perl code now can delete, add or modifiy HTTP header tags.

But be careful: because **<?!HTTPHEADER>** is a **preprocessor** command, the position of the **<?!HTTPHEADER>** command inside your CIPP program (even if you use it inside an Include), does not indicate the time, on which your HTTP header code is executed.

CIPP inserts the code you write in the **<?!HTTPHEADER>** block at the top of the generated CGI code, so it is executed before any other code you wrote in you CIPP program or Include, because the HTTP header must appear before any content.

So it is not possible to access any lexically scoped variables declared outside the **<?!HTTPHEADER>** block within the block. Usually you statically add or delete HTTP header fields. Your code may depend on CGI environment variables, or on a result of a SQL query, but that's it. If you want to access configuration variables, you must use the **<?CONFIG>** command inside your **<?!HTTPHEADER>** block.

Note

This command is not implemented for Apache::CIPP and CGI::CIPP environments, but you can use it with new.spirit .

Parameter

VAR

The actual HTTP header will be assigned to this variable, as a reference to a hash. This keys of this hash are the HTTP header tags.

MY

If you set the **MY** switch the created variable will be declared using 'my'. Its scope reaches to the end of the **<?!HTTPHEADER>** block .

Example

A HTTP header is created, which tells proxies how long they may cache the content of the produces HTML page.

```
<?!HTTPHEADER MY VAR="$http">
  # delete a Pragma Tag (may be defined
  # globally in a new.spirit environment)
  delete $http->{Pragma};

  # read a global config
  <?CONFIG NAME="x.global">

  # get cache time
  my $cache_time = $global::cachable_time || 1200;

  # set Cache-Control header tag
  $http->{'Cache-Control'} =
      "max-age=$cache_time, public";
<?!/HTTPHEADER>
```

IF

Type

Control Structure

Syntax

```
<?IF COND=condition >
...
[ <?ELSIF COND=condition > ]
...
[ <?ELSE> ]
...
<?/IF>
```

Description

The **<?IF>** command executes the enclosed block if the condition is true. **<?ELSE>** and **<?ELSIF>** can be used inside an **<?IF>** block in the common manner.

Parameter

COND
This takes a Perl condition. If this condition is true, the code inside the **<?IF>** block is executed.

Example

Only Larry gets a greeting message here.

```
<?IF COND="$name eq 'Larry'">
  Hi Larry!
<?/IF>
```

IMG

Type

HTML Tag Replacement

Syntax

```
<?IMG SRC=image_file
      [ additional_<IMG>_parameters ... ] >
```

Description

A HTML <IMG> Tag will be generated, whoms **SRC** option points to the appropriate image URL.

Parameter

SRC
This is the name of the image, expected as an URL in CGI::CIPP or Apache::CIPP environments and in dot-separated object notation in a new.spirit environment.

additional_<IMG>_parameters
All additional parameters are taken without changes into the produced <**IMG**> tag.

Example

In a new.spirit environment we produce a image link to another page, setting the border to 0.

```
<?A HREF="x.main.menu">
<?IMG SRC="x.images.logo" BORDER=0>
<?/A>
```

In CGI::CIPP or Apache::CIPP environment we provide an URL instead of a dot-separated object name.

```
<?A HREF="/main/menu.cgi">
<?IMG SRC="/images/logo.jpg" BORDER=0>
<?/A>
```

INCINTERFACE

Type

Interface

Syntax

```
<?INCINTERFACE [ INPUT=list_of_variables ]
               [ OPTIONAL=list_of_variables
               [ NOQUOTE=list_of_variables ]
               [ OUTPUT=list_of_variables ] >
```

Description

Use this command to declare an interface for an Include file. You can use this inside the Include file. In order to declare the interface of a CGI file this, use the **<?INTERFACE>** command.

You can declare mandatory and optional parameters. Parameters are always identified by name, not by position like in many programming languages. You can pass all types of Perl variables (scalars, arrays and hashes, also references). Also you can specify output parameters, which are passed back to the caller. Even these parameters are named, which requires some getting used to for most people. However it is **very** useful. :)

All input parameters declared this way are visible as the appropriate variables inside the Include file. They are always declared with **my** to prevent name clashes with other parts of the program.

Parameter

All parameters of **<?INCINTERFACE>** expect a comma separated list of variables. All Perl variable types are supported: scalars (**$**), arrays (**@**)and hashes (**%**). Whitespaces are ignored. Read the note beneath the **NOQUOTE** section about passing non scalar values to an Include.

Note: You have to use lower case variable names, because the **<?INCLUDE>** command converts all variable names to lower case.

INPUT

This parameters takes the list of variables the caller **must** provide in his **<?INCLUDE>** command (mandatory parameters).

OPTIONAL

The variables listed here are optional input parameters. They are always declared with **my** and visible inside the Include, but are set to **undef**, if the caller ommits them.

OUTPUT

If you want your Include to pass values back to the caller, list the appropriate variables here. This variables are declared with **my**. Set them everywhere in your Include, they will be passed back automatically.

Note: the name of the variable receiving the output from the include must be different from the name of the output parameter. This is due to restrictions of the internal implementation.

NOQUOTE

By default all input parameters are defined by assigning the given value using double quotes. This means it is possible to pass either string constants or string expressions to the Include, which are interpreted at runtime, in the same manner. Often this is the behaviour you expect.

You have to list input (no output) parameters in the **NOQUOTE** parameter if you want them to be interpreted as a real Perl expression, and not in the string context (e.g. **$i+1** will result in a string containing the value of **$i** concatenated with **+1** in a string context, but in an incremented **$i** otherwise).

Note: Also you have to list all non-scalar and reference input parameters here, because array, hash and reference variables are also computed inside a string context by default, and this is usually **not** what you expect.

Note: Maybe this will change in future. Listing array and hash parameters in **NOQUOTE** will be optional, the default behaviour for those variables will change, so that they are not computed in string context by default.

Notes

The **<?INCINTERFACE>** command may occur several times inside one Include file. The position inside the source code does not matter. All declarations will be added to an interface accordingly.

If you ommit a **<?INCINTERFACE>** command inside your Include, its interface is empty. That means, you cannot pass any parameters to it. If you try so this will result in an error message at CIPP compile time.

Example

This example declares an interface, expecting some scalars and an array. Note the usage of **NOQUOTE** for the array input parameter. The Include also returns a scalar and an array parameter.

```
<?INCINTERFACE INPUT="$firstname, $lastname"
               OPTIONAL="@id"
               OUTPUT="$scalar, @list"
               NOQUOTE="@id">
...
<?PERL>
  $scalar="returning a scalar";
  @list= ("returning", "a", "list");
<?/PERL>
```

The caller may use this **<?INCLUDE>** command. Note that all input parameter names are converted to lower case.

```
<?INCLUDE NAME="/include/test.inc"
          FIRSTNAME="Larry"
          lastname="Wall"
          ID="(5,4,3)"
          MY
          $s=SCALAR
          @l=LIST>
```

INCLUDE

Type

Import

Syntax

```
<?INCLUDE NAME=include_name
        [ input_parameter_1=Wert1 ... ]
        [ MY ]
        [ variable_1=output_parameter_1 ... ] >
```

Description

Use Includes to divide your project into reusable pieces of code. Includes are defined in separate files. They have a well defined interface due to the **<?INCINTERFACE>** command. CIPP performs parameter checking for you and complain about unknown or missing parameters.

The Include file code will be inserted at the same position you write **<?INCLUDE>**, inside of a Perl block. Due to this variables declared inside the Include are not valid outside.

Please refer to the **<?INCINTERFACE>** chapter to see how parameters are processed by an Include.

Parameter

NAME
This is the name of the Include file, expected as an URL in CGI::CIPP or Apache::CIPP environments and in dot-separated object notation in a new.spirit environment.

INPUT-PARAMETERS
You can pass parameters to the Include using the usual **PARAMETER=VALUE** notation. Note that parameter names are converted to lower case. For more details about Include input parameters refer to the appropriate section of the **<?INCINTERFACE>** chapter.

OUTPUT-PARAMETERS
You can recieve parameters from the Include using the notation

{$@%}variable=output_parameter

Note that the name of the output parameters are automatically converted to lower case. Note also that the caller must not use the same name like the output parameter for the local variable which recieves the output parameter. That means for the above notation that **variable** must be different from **output_parameter**, ignoring the case.

For more details about Include output parameters refer to the appropriate section of the **<?INCINTERFACE>** chapter.

MY

If you set the **MY** switch all created output parameter variables will be declared using 'my'. Their scope reaches to the end of the block which surrounds the **<?INCLUDE>** command.

Important note

The actual CIPP implementation does **really** include the Include code at the position where the **<?INCLUDE>** command occurs. This affects variable scoping. All variables visible at the callers source code where you write the **<?INCLUDE>** command are also visible inside your Include. So you can use these variables, although you never declared them inside your Include. Use of this feature is discouraged, in fact you should avoid the usage of variables you did not declared in your scope.

Short notation

In a new.spirit environment the **<?INCLUDE>** command can be abbreviated in the following manner:

```
<?include_name
    [ input_parameter_1=Wert1 ... ]
    [ MY ]
    [ variable_1=output_parameter_1 ... ] >
```

Example

See example of **<?INCINTERFACE>**.

INPUT

Type

HTML Tag Replacement

Syntax

```
<?INPUT [ NAME=parameter_name ]
        [ VALUE=parameter_value ]
        [ SRC=image_object ]
        [ TYPE=input_type ] [ STICKY[=sticky_var] ]
        [ additional_<INPUT>_parameters ... ] >
```

Description

This generates a HTML <INPUT> tag where the content of the VALUE option is escaped to prevent HTML syntax clashes. In case of TYPE="radio" or TYPE="checkbox" in conjunction with the STICKY Option, the state of the input widget will be preserved.

Parameter

NAME
The name of the input widget.

VALUE
This is the VALUE of the corresponding <INPUT> tag. Its content will be escaped.

SRC
This is the name of the image, expected as an URL in CGI::CIPP or Apache::CIPP environments and in dot-separated object notation in a new.spirit environment.

TYPE
Only the TYPEs „radio" and „checkbox" are specially handled when the STICKY option is also given.

STICKY

If this option is set and the TYPE of the input widget is eiterh „radio“ or „checkbox“ CIPP will generate the CHECKED option automatically, if the value of the corresponding Perl variable (which is $*parameter_name* for TYPE=“radio“ and @*parameter_name* for TYPE=“checkbox“) equals to the VALUE of this widget. If you assign a value to the STICKY option, this will be taken as the Perl variable for checking the state of the widget. But the default behaviour of deriving the name from the NAME option will fit most cases.

additional_<INPUT>_parameters

All additional parameters are taken without changes into the generated <**INPUT**> tag.

Note

If you use the STICKY feature in conjuncion with checkboxes, please note that the internal implementation may be ineffective, if you handle large checkbox groups. This is due the internal representation of the checkbox values as a list, so a grep is neccesary to check, wheter a checkbox is checked or not. If you feel uncomfortable about that, use a classic HTML <INPUT> tag, maybe with a loop around it, and check state of the checkboxes using a hash.

Example

We generate two HTML input fields, a simple text and a password field, both initialized with some values. Also two checkboxes are generated, using the STICKY feature to initalize their state genericly.

```
<?VAR MY NAME=$username>larry<?/VAR
<?VAR MY NAME=$password>this is my “password“<?/VAR>
<?INPUT TYPE=TEXT SIZE=40 VALUE=$username>
<?INPUT TYPE=PASSWORD SIZE=40 VALUE=$password>

<?VAR MY NAME=$check>42<?/VAR>
<?INPUT TYPE=CHECKBOX NAME=“check“ VALUE=“42“
        STICKY> 42
<?INPUT TYPE=CHECKBOX NAME=“check“ VALUE=“43“
        STICKY> 43
```

This will produce the following HTML code:

```
<INPUT TYPE=TEXT SIZE=40 VALUE=“larry“>
<INPUT TYPE=TEXT SIZE=40
       VALUE=“this ist my &quot;password&quot;“>

<INPUT TYPE=CHECKBOX NAME=“check“ VALUE=“42“
       CHECKED>
<INPUT TYPE=CHECKBOX NAME=“check“ VALUE=“43“>
```

INTERFACE

Type

Interface

Syntax

```
<?INTERFACE [ INPUT=list_of_variables ]
            [ OPTIONAL=list_of_variables ] >
```

Description

This command declares the interface of a CGI program. You can declare mandatory and optional parameters. Parameters are always identified by their name. You can recieve scalar and array parameters.

All input parameters declared this way are visible as the appropriate variables inside the CGI program. They are always declared with **my** to prevent name clashes with other parts of the program.

Using **<?INTERFACE>** is optional, if you are not in 'use strict' mode. If you ommit **<?INTERFACE>** all actual parameters are passed to your program, no parameter checking is done in this case. But it is strongly recommended to use **<?INTERFACE>** because CIPP checks the consistency of your CGI calls at runtime.

If you are in 'use strict' mode (which is the default), using **<?INTERFACE>** is mandatory, because one cannot create lexical variables at runtime. They must be declared in this manner, so CIPP can add the appropriate decalaration statements to the generated source code.

Parameter

All parameters of **<?INTERFACE>** expect a comma separated list of variables. Scalars (**$**) and arrays (**@**) are supported. Whitespaces are ignored.

Note: It is recommended that you use lower case variable names for your CGI interfaces, because some CIPP commands for generating URLs (e.g. **<?GETURL>**) convert parameter names to lower case.

INPUT

This parameters takes the list of variables the caller **must** pass to the CGI program.

OPTIONAL

The variables listed here are optional input parameters. They are always declared with **my** and visible inside the program, but are set to **undef**, if the caller ommits them.

Notes

The **<?INTERFACE>** command may occur several times inside a CGI program, the position inside the source code does not matter. All declarations will be added to an interface accordingly.

Example

We specify an interface for two scalars and an array.

```
<?INTERFACE INPUT="$firstname, $lastname"
            OPTIONAL="@id">
```

A HTML form which adresses this CGI program may look like this (assuming we are in a CGI::CIPP or Apache::CIPP environment).

```
<?VAR MY NAME="@id" NOQUOTE>(1,2,3,4)<?/VAR>

<?FORM ACTION="/user/save.cgi">
  <?HIDDENFIELDS PARAMS="@id">
  <P>firstname:
  <?INPUT TYPE=TEXT NAME=firstname>
  <P>lastname:
  <?INPUT TYPE=TEXT NAME=lastname>
<?/FORM>
```

LOG

Type

Exception Handling

Syntax

```
<?LOG MSG=error_message
      [ TYPE=type_of_message ]
      [ FILENAME=special_logfile ]
      [ THROW=exception ] >
```

Description

The **<?LOG>** command adds a line to the project specific logfile, if no other filename is specified. In new.spirit environments the default filename of the logfile is **prod/log/cipp.log**. In CGI::CIPP and Apache::CIPP environments messages are written to **/tmp/cipp.log** (**c:\tmp\cipp.log** under Win32) by default.

Log file entries contain a timestamp, client IP adress, a message type and the message itself.

Parameter

MSG
This is the message.

TYPE
You can use the **TYPE** parameter to speficy a special type for this message. This is simply a string. You can use this feature to ease logfile analysis.

FILENAME
If you want to add this message to a special logfile you pass the full path of this file with **FILENAME**.

THROW
With this parameter you can provide a user defined exception to be thrown on failure. The default exception thrown by this statement is **log**.

An exception will be thrown, if the log file is not writable or the path is not reachable.

Example

If the variable $error is set a simple entry will be added to the default logfile.

```
<?IF COND="$error != 0">
  <?LOG MSG="internal error: $error">
<?/IF>
```

The error message "error in SQL statement" is added to the special logfile with the path **`/tmp/my.log`**. This entry is marked with the special type **`dberror`**. If this file is not writable an exception called **`fileio`** is thrown.

```
<?LOG MSG="error in SQL statement"
      TYPE="dberror"
      FILE="/tmp/my.log"
      THROW="fileio">
```

MODULE

Type

Import

Syntax

```
<?MODULE NAME=cipp_perl_module >
  ...
<?/MODULE>
```

Description

With this command you define a CIPP Perl Module. This works currently in a new.spirit environment only.

The generated Perl code will be installed in the project specific lib/ folder and can be imported with the <?REQUIRE> command. Don't <?USE> for CIPP Perl modules, because <?REQUIRE> does some database initialization.

Parameter

NAME

This is the name of the module you want to use. Nested module names are delimited by **::**.

It is not possible to use a variable or expression for **NAME**, you must always use a literal string here.

Example

```
<?MODULE NAME=“Test::Module“>

<?SUB NAME=“new“>
  <?PERL>
    my $class = shift;
    return bless {
       foo => 1,
    }, $class;
  <?/PERL>
<?/SUB>

<?SUB NAME=“print_foo“>
  <?PERL>
    my $self = shift;
    print $self->{foo}, „<p>\n“;
  <?/PERL>
<?/SUB>

<?/MODULE>
```

MY

Type

Variables and Scoping

Syntax

```
<?MY [ VAR=list_of_variables ]
     variable_1 ... variable_N >
```

Description

This command declares private variables, using the Perl command **my** internally. Their scope reaches to the end of the block which surrounds the **<?MY>** command, for example only inside a **<?IF>** block.

All types of Perl variables (Scalars, Arrays and Hashes) can be declared this way.

If you want to initialize the variables with a value you must use the **<?VAR>** command or Perl commands directly. **<?MY>** only declares variables. Their initial value is **undef**.

Parameter

VAR
This parameter takes a comma separated list of variable names, that should be declared. With this option it is possible to declare variables which are not in lower case.

variable_1..variable_N
You can place additionel variables everywhere inside the **<?MY>** command. This variables are always declared in lower case notation.

Note:

If you need a new variable for another CIPP command, you can most often use the **MY** switch of that command, which declares the variable for you. This saves you one additional CIPP command and makes your code more readable.

Example

See **<?BLOCK>**

OPTION

Type

HTML Tag Replacement

Syntax

```
<?OPTION [ VALUE=parameter_value ]
         [ additional_<OPTION>_parameters ... ] >
...
<?/OPTION>
```

Description

This command generates a HTML <OPTION> tag, where the text inside the <OPTION> block is HTML escaped and the VALUE is quoted. The usage of the <?OPTION> command outside of a <?SELECT> block is forbidden. If the surrounding <?SELECT> command has its STICKY option set, the select state of the options are preserved (see <?SELECT> for more information about the STICKY feature).

Parameter

VALUE
This is the VALUE of the generated <OPTION> tag. Its content will be escaped.

additional_<OPTION>_parameters
All additional parameters are taken over without changes into the produced <**OPTION**> tag.

Example

See the description of the <?SELECT> command for a complete example.
.

PERL

Type

Control Structure

Syntax

```
<?PERL [ COND=condition ] >
...
<?/PERL>
```

Description

With this command you open a block with pure Perl commands. You may place any valid Perl code inside this block.

You may use the Perl **print** statement to produce HTML code (or whatever output you want) for the client.

At the moment, there are only two CIPP commands which are actually supported inside a **<?PERL>** block: **<?INCLUDE>** and **<?SQL>**. Support of more commands will be added in the future.

Parameter

COND

If you set the **COND** parameter, the Perl block is only executed, if the given condition is true.

Example

All occurences of the string 'nt' in the scalar variable $str will be replaced by 'no thanks'. The result will be printed to the client.

```
<?PERL>
  $text =~ s/nt/no thanks/g;
  print $text;
<?/PERL>
```

If this list contains some elements a string based on the list is generated.

```
<?PERL COND="scalar(@list) != 0">
  my ($string, $element);
  foreach $element ( @list ) {
    $string .= $element;
  }
  print $string;
<?/PERL>
# OK, its easier to use 'join', but it's
# only an example... :-)
```

.

!PROFILE

Type

Preprocessor Command

Syntax

```
<?!PROFILE { ON | OFF }
           [ DEEP ] >
```

Description

This preprocessor command controls the generation of profiling code. This feature is currently experimental, the syntax of the <?!PROFILE> command may change in future.

If you switch profiling on, CIPP will generate profile code for the rest of the file, respectively until a <?!PROFILE OFF> command occurs. Switching profiling at runtime is not possible, because the <?!PROFILE> command takes effect on the preprocessor.

Currently two tasks are profiled: SQL statements and Include executions. If profiling is switched on, you'll get a line on STDERR for every executed SQL and Include command, which contains the corresponding execution time. You need the Perl module Time::HiRes installed on your system if you want to use profiling.

Parameter

ON | OFF
Switch profiling either on or off.

DEEP
If you set the DEEP option, the content all processed Includes will be profiled, too. Otherwise only the document itself, where the <?!PROFILE> command stands, will be profiled.

Note that the DEEP switch can produce lots of output.

Example

The following SQL Statement and Include will be profiled.

```
<?!PROFILE ON>
<?SQL SQL="select foo, bla
           from   bar
           where  baz=?"
      PARAMS="$baz"
      MY VAR="$foo, $bla">
  $foo $bla<br>
<?/SQL>

<?INCLUDE NAME="/foo/bar.inc">

<?!/PROFILE OFF>

# no profiling here
<?INCLUDE NAME="/bar/foo.inc">
```

Something like this will appear on STDERR and thus in your webserver error log:

```
PROFILE 42421 START
PROFILE 42421 SQL        select foo, baz   0.0178
PROFILE 42421 INCLUDE    /foo/bar.inc      0.0020
PROFILE 42421 STOP
```

The 42421 is the PID of the serving process, so you can differ between outputs of several processes. You see the head of each SQL statement and the name of an Include, followed by the execution time in seconds.

You can use the PROFILE option of the <?SQL> command to replace the output of the SQL statement with a user defined label. See <?SQL> for details.

REQUIRE

Type

Import

Syntax

```
<?REQUIRE NAME="cipp_perl_module" >
```

Description

This command imports a module which was created with new.spirit in conjunction with the <?MODULE> command. You can't import other Perl modules, because <?REQUIRE> executes CIPP specific initialization code to establish database connections, if they're needed by the module.

<?REQUIRE> uses internally the Perl command 'require' to import the module. So CIPP Perl Modules are unable to export symbols to the callers namespace. You have to fully quallify function names, or write OO style modules.

Parameter

NAME

This is the name of the module you want to use. Nested module names are delimited by **::**. This is the name of the module you provided with the <?MODULE> command.

You may also place a scalar variable here, which contains the name of the module. So it is possible to load modules dynamically at runtime.

Example

```
# refer to the description of <?MODULE> to see
# the implementation of the Test::Module module.
<?REQUIRE NAME="Test::Module">

<?PERL>
  my $t = new Test::Module;
  $t->print_foo;
<?/PERL>
```

ROLLBACK

Type

SQL

Syntax

```
<?ROLLBACK [ DB=database_name ]
           [ DBH=database_handle ]
           [ THROW=exception ] >
```

Description

The **<?ROLLBACK>** command concludes the actual transaction and cancels all changes to the database.

Using **<?ROLLBACK>** in **<?AUTOCOMMIT ON>** mode is not possible.

If you are not in **<?AUTOCOMMIT ON>** mode a transaction begins with the first SQL statement and ends either with a **<?COMMIT>** or **<?ROLLBACK>** command.

Parameter

DB

This is the CIPP internal name of the database for this command. In CGI::CIPP or Apache::CIPP environment this name has to be defined in the appropriate global configuration. In a new.spirit environment this is the name of the database configuration object in dot-separated notation.

If DB is ommited the project default database is used.

DBH

Use this option to pass an existing DBI database handle, which should used for this SQL command. You can't use the **DBH** option in conjunction with **DB**.

THROW

With this parameter you can provide a user defined exception which should be thrown on failure. The default exception thrown by this statement is **rollback**.

If the underlying database is not capable of transactions (e.g. MySQL) execution of this command will throw an exception.

Example

We insert a row into a database table and rollback the change immediately. We throw a user defined exeption, if the rollback fails, maybe the database is not capable of transactions.

```
<?SQL SQL="insert into foo (num, str)
           values (42, 'bar');">
<?/SQL>
<?ROLLBACK THROW="ROLLBACK_Exception">
```

SAVEFILE

Type

Interface

Syntax

```
<?SAVEFILE FILENAME=server_side_filename
           VAR=upload_formular_variable
           [ SYMBOLIC ]
           [ THROW=exception ] >
```

Description

This command saves a file which was uploaded by a client in the webservers filesystem.

Parameter

FILENAME
This is the fully qualified filename where the file will be stored.

VAR
This is the identifier you used in the HTML form for the filename on client side, the value of the <INPUT NAME> parameter) .

SYMBOLIC
If this switch is set, VAR is the name of the variable which contains the <INPUT TYPE=FILE> identifier. Use this if you want to determine the name of the field at runtime.

THROW
With this parameter you can provide a user defined exception which should be thrown on failure. The default exception thrown by this statement is **savefile**.

Note

The client side file upload will only function proper if you set the encoding type of the HTML form to **ENCTYPE="multipart/form-data"**. Otherwise you will get a exception, that the file could not be fetched.

There is another quirk you should notice. The variable which corresponds to the <INPUT NAME> option in the file upload form is a GLOB reference (due to the internal implementation of the CGI module, which CIPP uses). That means, if you use that variable in string context you get the client side filename of the uploaded file. But also you can use the variable as a filehandle, to read data from the file (this is what **<?SAVEFILE>** does for you).

This GLOB thing is usually no problem, as long as you don't pass the variable as a binding parameter to a **<?SQL>** command (because you want to store the client side filename in the database). The DBI module (which CIPP uses for the database stuff) complains about passing GLOBS as binding parameters.

The solution is to create a new variable assigned from the value of the file upload variable enforced to be in string context using double quotes.

```
<?INTERFACE INPUT="$upfilename">
<?MY $client_filename>
<?PERL> $client_filename = "$upfilename" <?/PERL>
```

Example

First we provide a HTML form with the file upload field.

```
<?FORM METHOD="POST" ACTION="/image/save.cgi"
       ENCTYPE="multipart/form-data">
Fileupload:
<INPUT TYPE=FILE NAME="upfilename" SIZE=45><BR>
<INPUT TYPE="reset">
<INPUT TYPE="submit" NAME="submit" VALUE="Upload">
</FORM>
```

The **/image/save.cgi** program has the following code to store the file in the filesystem.

```
<?SAVEFILE FILENAME="/tmp/upload.tmp"
           VAR="upfilename"
           THROW=my_upload>
```

The same procedure using the **RUNTIME** parameter.

```
<?VAR MY=$field_name>upfilename<?/VAR>
<?SAVEFILE FILENAME="/tmp/upload.tmp"
           SYMBOLIC
           VAR="$field_name"
           THROW=upload>
```

SELECT

Type

HTML Tag Replacement

Syntax

```
<?SELECT [ NAME=parameter_name ]
         [ MULTIPLE ] [ STICKY ]
         [ additional_<SELECT>_parameters ... ] >
...
<?/SELECT>
```

Description

This command generates a selection widget providing preservation of the selection state (similar to the STICKY feature of the <?INPUT> command).

Parameter

NAME
The name of the formular widget.

MULTIPLE
If this is set, a multi selection list will be generated, instead of a single selection popup widget.

STICKY
If the STICKY option is set, the <?OPTION> commands inside the <?SELECT> block preserve their state in generating automatically a SELECTED option, if the corresponding entry was selected before. This is done in checking the value of the corresponding Perl variable (which is $*parameter_name* for a popup and @*parameter_name* for MULTIPLE selection list). If you assign a value to the STICKY option, this will be taken as the Perl variable for checking the state of the widget. But the default behaviour of deriving the name from the NAME option will fit most cases.

additional_<SELECT>_parameters
All additional parameters are taken over without changes into the produced <**SELECT**> tag.

Note

If you use the STICKY feature in conjuncion with a MULTIPLE selection list widget, please note that the internal implementation may be ineffective, if you handle large lists. This is due the internal representation of the list values as an array, so a grep is neccesary to check, wheter a list entry is selected or not. If you feel uncomfortable about that, use classic HTML <SELECT> and <OPTION> tags, maybe with a loop around it, and check state of the check-boxes using a hash.

Example

This is a complete CIPP program, which provides a mulitple selection list and preservers its state over subsequent executions of the program.

```
<?INCINTERFACE OPTIONAL="@list">

<?FORM ACTION="sticky.cgi">

<?SELECT NAME="list" MULTIPLE STICKY>
<?OPTION VALUE="1">value 1<?/OPTION>
<?OPTION VALUE="2">value 2<?/OPTION>
<?OPTION VALUE="3">value 3<?/OPTION>
<?/SELECT>

<?INPUT TYPE="submit" VALUE="send">

<?/FORM>
```

SQL

Type

SQL

Syntax

```
<?SQL SQL=sql_statement
      [ VAR=list_of_variables_for_the_result ]
      [ PARAMS=input_parameter ]
      [ WINSTART=start_row ]
      [ WINSIZE=number_of_rows_to_fetch ]
      [ RESULT=sql_return_code ]
      [ DB=database_name ]
      [ DBH=database_handle ]
      [ THROW=exception ] >
      [ MY ]
      [ PROFILE=profile_label ]
...
<?/SQL>
```

Description

Use the **<?SQL>** command to execute arbitrary SQL statements in a specific database. You can fetch results from a SELECT query, or simply execute INSERT, UPDATE or other SQL statements.

When you execute a SELECT query (resp. set the **VAR** parameter, see below) the code inside the **<?SQL>** block will be repeated for every row returned from the database.

Parameter

SQL

This takes the SQL statement to be executed. A trailing semicolon will be stripped off.

The statement may contain **?** placeholders. They will be replaced by the expressions listed in the **PARAMS** parameter. See the **PARAMS** section for details about placeholders.

This is an example of a simple insert without placeholders.

```
<?SQL SQL="insert into foo values (42, 'bar')">
<?/SQL>
```

VAR

If you set the **VAR** parameter, CIPP asumes that you execute a SQL statement which returns a result set (normally a SELECT statement).

The **VAR** parameter takes a list of scalar variables. Each variable corresponds to the according column of the result set, so the position of the variables inside the list is relevant.

You can use this variable inside the **<?SQL>** block to access the actual processed row of the result set. Below the **<?SQL>** block the variable contains the values of the last row fetched, even when they are implicitely declared via a **MY** switch.

This is an example of creating a simple HTML table out of an SQL result set.

```
<TABLE>
  <?SQL SQL="select num, str from foo"
        MY VAR="$n, $s">
    <TR>
      <TD>$n</TD>
      <TD>$s</TD>
    </TR>
  <?/SQL>
</TABLE>
```

PARAMS

All placeholders inside your SQL statement will be replaced with the values given in **PARAMS**. It expects a comma separated list (white spaces are ignored) of Perl expressions, normally variables (scalar or array), literals or constants. The Perl value **undef** will be translated to the SQL value **NULL**. The content of the first expression substitutes the first placeholder in the SQL string, etc.

Values of parameters are quoted, if necessary, before substitution. This is the main advantage of **PARAMS** in this context. (You could place the perl variables into the SQL statement as such, but you would have to use **<?DBQUOTE>** on them first. Or else.).

Beware that you cannot use placeholders to contain (parts of) SQL code. The **SQL** must contain the syntactically complete statement - placeholders can only contain values. (The main reason for this is that the SQL statement is parsed by most databases before the placeholders are substituted. See the DBI manpage for details about placeholders.)

Here are some examples which demonstate the usage of placeholders.

```
<?VAR MY NAME=$n>42<?/VAR>
<?VAR MY NAME=$s>Hello 'World'<?/VAR>
<?SQL SQL="insert into foo values (?, ?, ?)"
      PARAMS="$n, $s, time()">
<?/SQL>

<?VAR MY NAME=$where_num>42<?/VAR>
<?SQL SQL="select num,str from foo
           where num = ?"
      PARAMS="$where_num">
      MY VAR="$column_n, $column_s">
  n=$column_n s='$column_s'<BR>
<?/SQL>

<?SQL SQL="update foo
           set str=?
           where n=?"
      PARAMS="$s, $where_num">
<?/SQL>
```

WINSTART

If you want to process only a part of the result set you can specfiy the first row you want to see with the **WINSTART** parameter. All rows before the given **WINSTART** row will be fetched but ignored. Execution of the **<?SQL>** block begins with the **WINSTART** row.

The row count begins with 1.

Here is an example. The first 5 rows will be skipped.

```
<?SQL SQL="select num, str from foo"
      MY VAR="$n, $s"
      WINSTART=6
  n=$n s='$s'<BR>
<?/SQL>
```

WINSIZE

Set this parameter to specify the number of rows you want to process. You can combine this parameter with **WINSTART** to process a "window" of the result set.

This is an example of doing this (skipping 5 rows, processing 5 rows).

```
<?SQL SQL="select num, str from foo"
      MY VAR="$n, $s"
      WINSTART=6 WINSIZE=5
  n=$n s='$s'<BR>
<?/SQL>
```

RESULT

Some SQL statements return a scalar result value, e.g. the number of rows processed (e.b. UPDATE and DELETE). The variable placed here will take the SQL result code, if there is one.

Example:

```
<?SQL SQL="delete from foo where num=42"
      MY RESULT=$deleted>
<?/SQL>
Successfully deleted $deleted rows!
```

DB

This is the CIPP internal name of the database for this command. In CGI::CIPP or Apache::CIPP environment this name has to be defined in the appropriate global configuration. In a new.spirit environment this is the name of the database configuration object in dot-separated notation.

If DB is ommited the project default database is used.

DBH

Use this option to pass an existing DBI database handle, which should used for this SQL command. You can't use the **DBH** option in conjunction with **DB**.

THROW

With this parameter you can provide a user defined exception which should be thrown on failure. The default exception thrown by this statement is **sql**.

MY

If you set the **MY** switch all created variables will be declared using 'my'. Their scope reaches to the end of the block which surrounds the **<?SQL>** command.

PROFILE

Here you can define a profile label for this SQL statement. If you use the <?!PROFILE> command this label is printed out instead of the head of the SQL statement. See the chapter of <?!PROFILE> about details.

Breaking the <?SQL> loop

If you want to break the SQL loop of a select statement, simply use this Perl Code:

```
<?PERL>
  last SQL;
<?/PERL>
```

Example

Please refer to the examples in the parameter sections above.

SUB

Type

Control Structure

Syntax

```
<?SUB NAME=subroutine_name >
...
<?/SUB>
```

Description

This defines the **<?SUB>** block as a Perl subroutine. You may use any CIPP commands inside the block.

Generally Includes are the best way to create reusable modules with CIPP. But sometimes you need real Perl subroutines, e.g. if you want to do some OO programming.

Parameter

NAME

This is the name of the subroutine. Please refer to the perlsub manpage for details about Perl subroutines.

It is not possible to declare protoyped subroutines with **<?SUB>**.

Example

This is a subroutine to create a text input field in a specific layout.

```
<?SUB NAME=print_input_field>
  # Catch the input parameters
  <?MY $label $name $value>
  <?PERL>
    ($label, $name, $value) = @_;
  <?/PERL>

  # print the text field
  <P>
  <B>$label:</B><BR>
  <?INPUT TYPE=TEXT SIZE=40 NAME=$name VALUE=$value>
<?/SUB>
```

You may call this subroutine from every Perl context this way.

```
<?PERL>
  print_input_field (’Firstname’, ’firstname’,
                     ’Larry’);
  print_input_field (’Lastname’, ’surname’,
                     ’Wall’);
<?/PERL>
```

TEXTAREA

Type

HTML Tag Replacement

Syntax

```
<?TEXTAREA [ additional_<TEXTAREA>_parameters ... ]>
...
<?/TEXTAREA>
```

Description

This generates a HTML <TEXTAREA> tag, with a HTML quoted content to prevent from HTML syntax clashes.

Parameter

additional_<TEXTAREA>_parameters
There are no special parameters. All parameters you pass to **<?TEXTAREA>** are taken in without changes.

Example

This creates a <TEXTAREA> initialized with the content of the variable $fulltext.

```
<?VAR MY NAME=$fulltext><B>HTML Text</B><?/VAR>
<?TEXTAREA NAME=fulltext ROWS=10
COLS=80>$fulltext<?/TEXTAREA>
```

This leads to the following HTML code.

```
<TEXTAREA NAME=fulltext ROWS=10
COLS=80>&lt;B>HTML Text&lt;B></TEXTAREA>
```

THROW

Type

Exception Handling

Syntax

<?**THROW THROW**=*exception* [**MSG**=*message*] >

Description

This command throws an user specified exception.

Parameter

THROW
This is the exception identifier, a simple string. It is the criteria for the **<?CATCH>** command.

MSG
Optionally, you can pass a additional message for your exception, e.g. a error message you have got from a system call.

Example

We try to open a file and throw a exception if this fails.

```
<?MY $error>
<?PERL>
  open (INPUT, '/bar/foo') or $error=$!;
<?/PERL>

<?IF COND="$error">
  <?THROW THROW="open_file"
          MSG="file /bar/foo, $error">
<?/IF>
```

Note

If you want to throw a exception inside a Perl block you can do this with the Perl **die** function. The die argument must follow this convention:

identifier TAB **message**

This is the above example using this technique.

```
<?PERL>
  open (INPUT, ’/bar/foo’)
    or die “open_file\tfile /bar/foo, $!“;
<?/PERL>
```

TRY

Type

Exception Handling

Syntax

```
<?TRY >
...
<?/TRY >
```

Description

Normally your program exits with a general exception message if an error/exception occurs or is thrown explicitely. The general exception handler which is responsible for this behaviour is part of any program code which CIPP generates.

You can provide your own exception handling using the **<?TRY>** and **<?CATCH>** commands.

All exceptions thrown inside a **<?TRY>** block are caught. You can use a subsequent **<?CATCH>** block to process the exceptions to set up your own exception handling.

If you ommit the **<?CATCH>** block, nothing will happen. You never see something of the exception, it will be fully ignored and the program works on.

Example

We try to insert a row into a database table and write a log file entry with the error message, if the INSERT fails.

```
<?TRY>
  <?SQL SQL="insert into foo values (42, 'bar')">
  <?/SQL>
<?/TRY>

<?CATCH THROW="insert" MY MSGVAR="$msg">
  <?LOG MSG="unable to insert row, $msg"
        TYPE="database">
<?/CATCH>
```

URLENCODE

Type

URL and Form Handling

Syntax

```
<?URLENCODE VAR=unencoded_variable
            [ MY ] ENCVAR=encoded_variable >
```

Description

Use this command to URL encode the content of a scalar variable. Parameters passed via URL always have to be encoded this way, otherwise you risk syntax clashes.

Parameter

VAR
This is the variable you want to be encoded.

ENCVAR
The encoded result will be stored in this variable.

MY
If you set the **MY** switch the created variable will be declared using 'my'. Its scope reaches to the end of the block which surrounds the **<?URLENCODE>** command.

Example

In this example we link an external CGI script and pass the content of the variable $query after using **<?URLENCODE>** on it.

```
<?URLENCODE VAR=$query MY ENCVAR=$enc_query>
<A HREF="www.search.org?query=$enc_query">
find something
</A>
```

Hint: in CGI::CIPP and Apache::CIPP environments you also can use the **<?A>** command for doing this.

USE

Type

Import

Syntax

<?**USE NAME**=*perl_module* >

Description

With this command you can access the extensive Perl module library. You can access any Perl module which is installed on your system.

In a new.spirit environment you can place user defined modules in the **prod/lib** directory of your project, which is included in the library search path by default.

If you want to use a CIPP Module (generated with new.spirit and the <?MODULE> command), use <?REQUIRE> instead.

Parameter

NAME

This is the name of the module you want to use. Nested module names are delimited by **::**. This is exactly what the Perl **use** pragma expects (you guessed right, CIPP simply translates **<?USE>** to **use** :-).

It is not possible to use a variable or expression for **NAME**, you must always use a literal string here.

Example

The standard modules File::Path and Text::Wrap are imported to your program.

```
<?USE NAME="File::Path">
<?USE NAME="Text::Wrap">
```

VAR

Type

Variables and Scoping

Syntax

```
<?VAR NAME=variable
      [ MY ]
      [ DEFAULT=value ]
      [ NOQUOTE ]>
...
<?/VAR>
```

Description

This command defines and optionally declares a Perl variable of any type (scalar, array and hash). The value of the variable is derived from the content of the **<?VAR>** block. You can assign constants, string expressions and any Perl expressions this way.

It is not possible to nest the **<?VAR>** command or to use any CIPP command inside the **<?VAR>** block. The content of the **<?VAR>** block will be evaluated and assigned to the variable.

Parameter

NAME

This is the name of the variable. You must specify the full Perl variable here, including the **$**, **@** or **%** sign to indicate the type of the variable.

These are some examples for creating variables using **<?VAR>**.

```
<?VAR NAME=$skalar>a string<?/VAR>
<?VAR NAME=@liste>(1,2,3,4)<?/VAR>
<?VAR NAME=%hash>( 1 => 'a', 2 => 'b' )<?/VAR>
```

DEFAULT

If you set the **DEFAULT** parameter, this value will be assigned to the variable, if the variable is **actually** undef. In this case the content of the **<?VAR>** block will be ignored.

Setting the **DEFAULT** parameter is only supported for scalar variables.

You can use this feature to provide default values for input parameters this way.

```
<?VAR NAME=$event DEFAULT="show">$event<?/VAR>
```

Hint: you may think there must be a easier way of doing this. You are right. :-) We recommend you using this alternative code, the usage of **DEFAULT** is deprecated.

```
<?PERL>
  $event ||= 'show';
<?/PERL>
```

NOQUOTE

By default the variable is defined by assigning the given value using double quotes. This means it is possible to assign either string constants or string expressions to the variable without using extra quotes.

If you do not want the content of **<?VAR>** block to be evaluated in string context set the **NOQUOTE** switch. E.g., so it is possible to assign an integer expression to the variable.

This is an example of using **NOQUOTE** for an non string expression.

```
<?VAR MY NAME=$element_cnt NOQUOTE>
  scalar(@liste)
<?/VAR>
```

MY

If you set the **MY** switch the created variable will be declared using 'my'. Its scope reaches to the end of the block which surrounds the **<?VAR>** command.

Example

Please refer to the examples in the parameter sections above.

WHILE

Type

Control Structure

Syntax

```
<?WHILE COND=condition >
...
<?/WHILE>
```

Description

This realizes a loop, where the condition is checked first before entering the loop code.

Parameter

COND

As long as this Perl condition is true, the **<?WHILE>** block will be repeated.

Example

This creates a HTML table out of an array using **<?WHILE>** to iterate over the two arrays @firstname and @lastname, assuming that they are of identical size.

```
<TABLE>
<?VAR MY NAME=$i>0<?/VAR>
<?WHILE COND="$i++ < scalar(@lastname)">
  <TR>
    <TD>$lastname[$i]</TD>
    <TD>$firstname[$i]</TD>
  </TR>
<?/WHILE>
</TABLE>
```