Use ActiveRecord In Your Ruby Project

Posted by Panos M. (http://www.linkedin.com/in/panayotismatsinopoulos) on 06/Dec/2017 (08:38)

Use ActiveRecord in your Ruby project

ActiveRecord (https://github.com/rails/rails/tree/master/activerecord) is a gem that is part of Ruby on Rails (http://rubyonrails.org/). It is the ORM (https://en.wikipedia.org/wiki/Object-relational_mapping), i.e. the library that maps our objects to tables. In other words, it is the Ruby library that allows us to use Ruby classes in order to access our data stored in an RDBMS, like MySQL or PostgreSQL.

Ruby on Rails and ActiveRecord dance together very nicely. But, what if we want to develop a Ruby application that is not a Web Ruby on Rails application. What if it is only a simply Ruby application that needs to access a backend RDBMS. Can we use ActiveRecord again?

Yes, we can. And this is what we are going to demonstrate in this blog post.

Initiate a Ruby Project

Let's start with initialization of our Ruby project. We are going to create a Ruby application that manages movies. Hence, let's call the project `movies_app`.

(1) I am using RVM (http://rvm.io/) to create a new gemset:

```
$ rvm use 2.4.2@movies_app --create
ruby-2.4.2 - #gemset created /Users/panayotismatsinopoulos/.rvm/gems/ruby-2.4.2@movies_app
ruby-2.4.2 - #generating movies_app wrappers.........
Using /Users/panayotismatsinopoulos/.rvm/gems/ruby-2.4.2 with gemset movies_app
$
```

(2) I then create the root folder of the project. Let's call it movies_app

```
$ mkdir movies_app
$ cd movies_app
movies_app $
```

(3) Then create the files that freeze the rvm gemset:

```
movies_app $ echo '2.4.2' > .ruby-version
movies_app $ echo 'movies_app' > .ruby-gemset
movies_app $
```

With the above in place, I am sure that whenever I cd to the root folder of this project, I will be using the correct gemset.

Install `bundler`

This is necessary in order for us to download the necessary gems.

```
movies_app $ gem install bundler --no-ri --no-rdoc
Fetching: bundler-1.16.0.gem (100%)
Successfully installed bundler-1.16.0
1 gem installed
movies_app $
```

Create `Gemfile`

We will now create the `Gemfile` and add the activerecord gem in. Also, we will add the gem standalone_migrations (https://github.com/thuss/standalone-migrations). This gem is very useful, because it allows us to execute `rake` tasks related to migrations, like we do with the Rails standard tasks.

```
# Gemfile
source 'https://rubygems.org'

gem 'activerecord'
gem 'standalone_migrations'
```

Install Gems

Having created the `Gemfile`, now let's proceed to the installation of the necessary gems. Run `bundle` and you will see the gems downloaded and installed.

```
movies_app $ bundle
Fetching gem metadata from https://rubygems.org/..............
Resolving dependencies...
Fetching rake 12.3.0
...
Fetching standalone_migrations 5.2.3
Installing standalone_migrations 5.2.3
Bundle complete! 2 Gemfile dependencies, 27 gems now installed.
Use `bundle info [gemname]` to see where a bundled gem is installed.
movies_app $
```

Decide On Your Database

Now, you will need to choose your database. Is it going to be SQLite3? Is it going to be PostgreSQL or MySQL? Other? I have PosgreSQL installed locally on my machine and that's why I have decided to go with PostgreSQL. This means that I need one more gem in my Gemfile. The pg (https://rubygems.org/gems/pg) gem.

This is the new version of my `Gemfile`:

```
# Gemfile
source 'https://rubygems.org'

gem 'activerecord'
gem 'standalone_migrations'
gem 'pg'
```

And then I do a `bundle` again.

```
movies_app $ bundle
...
Fetching pg 0.21.0
Installing pg 0.21.0 with native extensions
Using thor 0.20.0
Using railties 5.1.4
...
movies_app $
```

If you were to use SQLite3, then you would have to add the gem `sqlite3`. If you were to use MySQL, then you would have to add the gem `mysql2`.

Create Database Configuration File

With the gem for the database access in place, now lets create the database configuration file. This is going to be a file inside the folder `db` and having the name `config.yml`. Hence, create the file `db/config.yml` with the following content:

```
default: &default
  adapter: postgresql
  encoding: unicode
  pool: 5
  host: localhost

development:
  <<: *default
  database: movies_development

test: &test
  <<: *default
  database: movies_test

production:
```

The above `db/config.yml` file is an indicative file for the PostgreSQL database. You should adapt the content of this file according to your database server.

Create Rakefile

Now go ahead and create a `Rakefile` at the root folder of your project. This file needs to have the following content:

```
# Rakefile
#
require 'standalone_migrations'
StandaloneMigrations::Tasks.load_tasks
```

It will load all the tasks that will help you create and manage your migrations.

To double check that you have access to the tasks, run the following command:

```
movies_app $ bundle exec rake --tasks
...
```

You will get a long list of the tasks that you have available.

Create Database

The next step is that you create your local database:

```
movies_app $ bundle exec rake db:create
Created database 'movies_development'
Created database 'movies_test'
movies_app $
```

The above rake task created both a development and a test database. And it is the same rake command like we are used to from Rails.

Create First Migration

Now, let's create the first db migration. This is a little bit different to the way you generate migrations in Rails. In Rails, we use the `rails generate migration` command. Here, we are going to use a rake task:

```
movies_app $ bundle exec rake db:new_migration[create_movies]
      create  db/migrate/20171207051113_create_movies.rb
movies_app $
```

But it worked like the Rails case.

Edit Migration And Invoke

Let's edit the file `db/migrate/20171207051113_create_movies.rb`

```
class CreateMovies < ActiveRecord::Migration[5.1]
  def change
    create_table :movies do |t|
      t.string :title, null: false
      t.string :director, null: false

      t.timestamps
    end
  end
end
```

The above is a very simple db schema migration. It creates the table `movies` with two main columns and the columns for the timestamps.

Let's invoke the migration, like we do with Rails migrations:

```
movies_app $ bundle exec rake db:migrate
(in /Users/panayotismatsinopoulos/Documents/movies_app)
== 20171207051113 CreateMovies: migrating ===================================
-- create_table(:movies)
   -> 0.0063s
== 20171207051113 CreateMovies: migrated (0.0064s) ==========================

movies_app $
```

Nice! The migration has been executed successfully. And it output similar information like the one we get when we run migrations with Rails.

Confirm Table Creation

You can confirm the table creation, using your SQL command line interface. For PostgreSQL, I can do this:

```
movies_app $ psql -d movies_development -c "\d+ movies"
                                                  Table "public.movies"
   Column   |            Type             |                      Modifiers                       | Storage  | Sta
------------+-----------------------------+------------------------------------------------------+----------+----
 id         | bigint                      | not null default nextval('movies_id_seq'::regclass) | plain    |
 title      | character varying           | not null                                             | extended |
 director   | character varying           | not null                                             | extended |
 created_at | timestamp without time zone | not null                                             | plain    |
 updated_at | timestamp without time zone | not null                                             | plain    |
Indexes:
    "movies_pkey" PRIMARY KEY, btree (id)

movies_app $
```

Excellent!

Create a Model

Let's see now how we can create an ActiveRecord model that would allow us to manage movies. Create the file `app/models/movie.rb` with the following content:

```ruby
# app/models/movie.rb
#
class Movie < ActiveRecord::Base
  validates :title, presence: true, uniqueness: {case_insensitive: true}
  validates :director, presence: true
end
```

It is very simple. A class that derives from `ActiveRecord::Base`.

Our Main Application

Finally, let's create a main file that would use the `Movie` model to create a movie in the database. Create the file `app/main.rb` with the following content:

```ruby
require 'active_record'
require_relative './models/movie'

def db_configuration
  db_configuration_file = File.join(File.expand_path('..', __FILE__), '..', 'db', 'config.yml')
  YAML.load(File.read(db_configuration_file))
end

ActiveRecord::Base.establish_connection(db_configuration["development"])

print "Give me the title of the movie: "
title = gets.chomp

print "Give me the director of the movie: "
director = gets.chomp

title = Movie.new(title: title, director: director)
title.save!

puts "Number of movies in your database: #{Movie.count}"
puts "Bye!"
```

This is a very simple program that uses ActiveRecord to connect to your database and then create a Movie based on the data provided by the user. Pay attention to the lines 4 till 9. These are necessary in order for you to establish a database connection. It loads the database connection configuration data from the `db/config.yml` file and then picks up the `development` environment part and sends it to `ActiveRecord::Base.establish_connection` method. Needless to say, that this code needs to be adapted to read the environment part dynamically.

An instance of running this program is given below:

```
movies_app $ bundle exec ruby app/main.rb
Give me the title of the movie: Jurassic Park
Give me the director of the movie: Steven Spielberg
Number of movies in your database: 1
Bye!
movies_app $
```

Closing Note

The above is the bare minimum that would allow you to incorporate ActiveRecord into your Ruby (non-Rails) application. The help of the gem `standalone_migrations` is very valuable because it allows us to use the ActiveRecord Migrations API like we do with the Rails applications.

Please share your thoughts in the comments section below, as I always learn a lot from you.

Finally, I want to mention here that on our Full Stack Web Developer course (/full-stack-web-developer) we teach both Ruby and Ruby on Rails. This is a Mentor supported course that you pay-as-you-go. Your Mentor is assigned to you and evaluates your work and your progress, making sure that you improve on every step that you take.

 (https://www.techcareerbooster.com/blog/use-activerecord-in-your-ruby-project)
 (https://twitter.com/intent/tweet?text=Interesting+Content&url=https%3A%2F%2Fwww.techcareerbooster.com%2Fblog%2Fuse-activerecord-in-your-ruby-project&hashtags=careerbooster&via=techcareerboost&related=techcareerboost%2C+Tweet+about+TCB+courses)
 (https://www.linkedin.com/company/tech-career-booster)