PADRINO BOOK

Be quick. Be elegant. Use Padrino.

Matthias Günther

PadrinoBook

The Guide To Master The Elegant Ruby Web Framework

Matthias Günther

Contents

Chapter 1

Introduction and Setup

Why another book about how to develop an application (app) in Rails? But wait, this book should give you a basic introduction on how to develop a web app with Padrino. Padrino is “The Elegant Ruby Web Framework”. Padrino is based upon Sinatra, which is a simple Domain Specific Language for quickly creating web apps in Ruby. When writing Sinatra apps many developers miss some of the extra conveniences that Rails offers, this is where Padrino comes in as it provides many of these while still staying true to Sinatra’s philosophy of being simple and lightweight. In order to understand the mantra of the Padrino webpage: “Padrino is a full-stack ruby framework built upon Sinatra” you have to read on.

1.1 Motivation

Shamelessly I have to tell you that I’m learning Padrino through writing a book about instead of doing a blog post series about it. Besides I want to provide up-to-date documentation for Padrino which is at the moment scattered around the Padrino’s web page padrinorb.com.

Although Padrino borrows many ideas and techniques from it’s big brother Rails it aims to be more modular and allows you to interchange various components with considerable ease. You will see this when you will the creation of two different application we are going to build throughout the book.

1.1.1 Why Padrino

Nothing is enabled without explicit choice. You as a programmer know what database is best for your application, which Gems don't carry security issues. If you are honest to yourself you can only learn a framework by heart if you go and digg under the hood. Because Padrino is small and you can understand most of the source. There is no need for monkey-patching, almost everything can be changed via an API. Padrino is rack-friendly, a lot of techniques that are common to Ruby can be reused. Having a low stack frame makes it easier for debugging. The best Rails convenience parts like `I18n` and `active_support` are available for you.

Before going any further you may ask: Why should you care about learning and using another web framework? Because you want something that is *easy to use*, *simple to hack*, and *open to any contribution*. If you've done Rails before, you may reach the point where you can't see how things are solved in particular order. In other words: There are many layers between you and the core of you application. You want to have the freedom to chose which layers you want to use in your application. This freedoms comes with the help of the Sinatra framework.

Padrino adds the core values of Rails into Sinatra and gives you the following extras:

- `orm`: Choose which adapter you want for a new application.
- `multiple application support`: Split you application into small, more manageble-and-testable parts that are easier to maintain and to test.
- `admin interface`: Provides an easy way to view, search, and modify data in your application.

When you are starting a new project in Padrino only a few files are created and, when your taking a closer look at them, you will see what each part of the code does. Having less files means less code and that is easier to maintain. Less code means that your application will run faster.

With the ability to manage different applications, for example: for your blog, your image gallery, or your payment cycle; by separating your business logic, you can share data models, session information and the admin interface between them without duplicating code.

Remember: "**Be tiny. Be fast. Be a Padrino.**"

1.1.2 Pages using Padrino

Here is a rough list of pages using Padrino:

- Coca Cola Enterprises - Coca Cola's European bottling arm
- Maptia - A beautiful way to tell stories about places.
- Brainfeed - Back-end for iPad app that presents educational videos for kids.
- martianoids.com - System administration company at Spain.
- HRPartner - The go-to cloud HR software for small & medium-sized businesses.
- tokyo-project - A photo gallery showcasing pictures of Tokyo
- HOF Studios Website - HOF Studios specializes in game development for PC and mobile platforms.

Even more open-source application, web libraries as well as other pages can be found on padrinorb page.

1.2 Tools and Knowledge

I won't tell you which operating system you should use - there is an interesting discussion on hackernews. I'll leave it free for the reader of this book which to use, because you are reading this book to learn Padrino.

To actually see a running padrino app, you need a web browser of your choice. For writing the application, you can either use an Integrated Development Environment (IDE) or with a plain text editor.

Nowadays there are a bunch of Integrated Development Environments (IDEs) out there:

- RubyMine by JetBrains - commercial, available for all platforms
- Eclipse Dynamic Languages Toolkit - free, available for all platforms

Here is a list of plain text editors which are a popular choice among Ruby developers:

- Emacs - free, available for all platforms.
- Gedit - free, available for Linux and Windows.
- Notepad++ - free, available only for Windows.
- SublimeText - commercial, available for all platforms.
- Textmate - commercial, available only for Mac.
- Vim - free, available for all platforms.

All tools have their strengths and weaknesses. Try to find the software that works best for you. The main goal is that you comfortable because you will spend a lot of time with it.

1.2.1 Installing Ruby With rbenv

Instead of using the build-in software package for Ruby of your operating system, we will use rbenv which lets you switch between multiple versions of Ruby.

First, we need to use git to get the current version of rbenv:

```
$ cd $HOME
$ git clone git://github.com/sstephenson/rbenv.git .rbenv
```

In case you shouldn't want to use git, you can also download the latest version as a zip file from GitHub.

You need to add the directory that contains rbenv to your `$PATH` environment variable. If you are on Mac, you have to replace `.bashrc` with `.bash_profile` in all of the following commands):

```
$ echo 'export PATH="$HOME/.rbenv/bin:$PATH"' >> ~/.bashrc
```

To enable auto completion for `rbenv` commands, we need to perform the following command:

```
$ echo 'eval "$(rbenv init -)"' >> ~/.bashrc
```

Next, we need to restart our shell to enable the last changes:

```
$ exec $SHELL
```

Basically, there are two ways to install different versions of Ruby: You can compile Ruby on your own and try to manage the versions and gems on your own, or you use a tool that helps you.

ruby-build

Because we don't want to download and compile different Ruby versions on our own, we will use the ruby-build plugin for rbenv:

```
$ mkdir ~/.rbenv/plugins
$ cd ~/.rbenv/plugins
$ git clone git://github.com/sstephenson/ruby-build.git
```

If you now run `rbenv install` you can see all the different Ruby version you can install and use for different Ruby projects. We are going to install `ruby 2.4.1`:

```
$ rbenv install 2.4.1
```

This command will take a couple of minutes, so it's best to grab a Raider, which is now known as Twix. After everything runs fine, you have to run `rbenv rehash` to rebuild the internal rbenv libraries. The last step is to make Ruby 2.4.1 the current executable on your machine:

```
$ rbenv global 2.4.1
```

Check that the correct executable is active by exexuting `ruby -v`. The output should look like:

```
$ 2.4.1 (set by /home/.rbenv/versions)
```

Now you are a ready to hack on with Padrino!

1.2.2 Ruby Knowledge

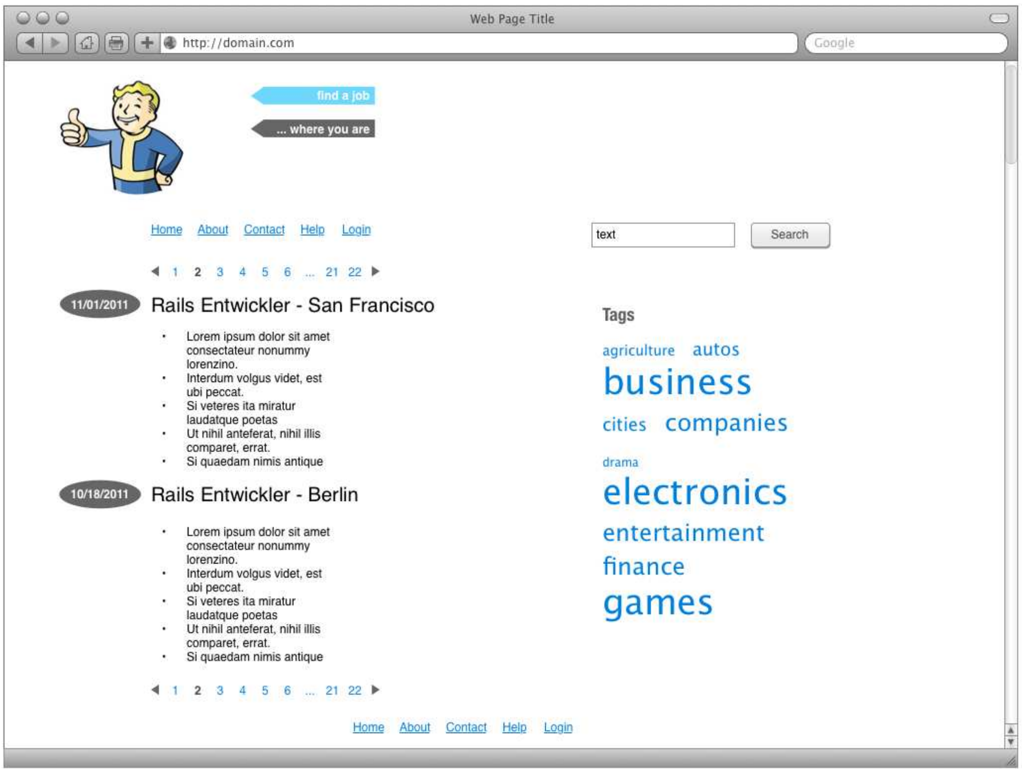
For any non-Ruby people, I strongly advise you to check out one of these books and learn the basics of Ruby before continuing here.

- Programming Ruby - the standard book on Ruby.
- Poignant Guide to Ruby - written by why the lucky stiff in an entertaining and educational way.

In this book, I assume readers having Ruby knowledge and will not be explaining every last detail. I will explain Padrino-specific coding techniques and how to get most parts under test.

1.3 Hello Padrino

The basic layout of our application is displayed on the following image application:



It is possible that you know this section from several tutorials, which makes you even more comfortable with your first program.

Now, get your hands dirty and start coding.

First of all we need to install the padrino gem. We are using the last stable version of Padrino (during the release of this book it is version 0.14.0.1). Execute this command.

```
$ gem install padrino
```

This will install all necessary dependencies and gets you ready to start. Now we will generate a fresh new Padrino project:

```
$ padrino generate project hello-padrino
```

Let's go through each part of this command:

- `padrino generate`:[1] Tells Padrino to execute the generator with the specified options. The options can be used to create other **components** for your app, like a **mailing system** or an **admin panel** to manage your database entries. We will handle these things in a future chapter. A shortcut for generate is `g` which we will use in all following examples.
- `project`: Tells Padrino to generate a new app.
- `hello-padrino`: The name of the new app, which is also the directory name.

The console output should look like the following:

```
  create
  create  .gitignore
  create  config.ru
  create  config/apps.rb
  create  config/boot.rb
  create  public/favicon.ico
  create  public/images
  create  public/javascripts
  create  public/stylesheets
  create  .components
  create  app
  create  app/app.rb
  create  app/controllers
  create  app/helpers
  create  app/views
  create  app/views/layouts
  append  config/apps.rb
  create  Gemfile
  create  Rakefile
  create  exe/hello-padrino
  create  tmp
  create  tmp/.keep
```

[1]You can also use `padrino g` or `padrino-gen` for the `generate` command, which will be used in the rest of this book

```
  create  log
  create  log/.keep
skipping  orm component...
skipping  test component...
skipping  mock component...
skipping  script component...
skipping  renderer component...
skipping  stylesheet component...
identical  .components
   force  .components
   force  .components

=================================================================
hello-padrino is ready for development!
=================================================================
$ cd ./hello-padrino
$ bundle --binstubs
=================================================================
```

The last line in the console output tells you the next steps you have to perform. Before we start coding our app, we need some sort of package management for Ruby gems.

Ruby has a nice package manager called bundler which installs all necessary gems in the versions you would like to have for your project. Other developers know now how to work with your project even after years. The Gemfile declares the gems that you want to install. Bundler takes the content of the Gemfile and will install every package declared in this file.

To install bundler 1.14.6, execute the following command and check the console output:

```
$ gem install bundler
```

Now we have everything we need to run the `bundle` command and install our dependencies:

```
$ cd hello-padrino
$ bundle
  Fetching gem metadata from https://rubygems.org/.........
```

Let's open the file **app/app.rb** (think of it as the root controller of your app) and insert the following code before the last **end**:

```
# app/app.rb

module HelloWorld
  class App < Padrino::Application

    get "/" do
      "Hello Padrino!"
    end

  end
end
```

Now run the app with:

```
$ bundle exec padrino start
```

Instead of writing **start**, we can also use the **s** alias. Now, fire up your browser with the URL http://localhost:3000 and see the **Hello World** Greeting being printed.

Congratulations, you've built your first Padrino app!

1.3.1 Directory Structure of Padrino

Navigating through the various parts of a project is essential. Thus we will go through the basic file structure of the *hello-padrino* project. The app consists of the following parts:

```
|-- app
|   |-- app.rb
|   |-- controllers
|   |-- helpers
|   `-- views
|       `-- layouts
|-- bin
|-- config
|   |-- apps.rb
```

```
|   |-- boot.rb
|   `-- database.rb
|-- config.ru
|-- exe
|   |-- hello-padrino
|-- Gemfile
|-- Gemfile.lock
|-- public
|   |-- favicon.ico
|   |-- images
|   |-- javascripts
|   `-- stylesheets
|-- Rakefile
`-- tmp
```

We will go through each part.

- **app**: Contains the "executable" files of your project, along with the controllers, helpers, and views of your app.
  - **app.rb**: The primary configuration file of your application. Here you can enable or disable various options like observers, your mail settings, specify the location of your assets directory, enable sessions, and other options.
  - **controller**: The controllers make the model data available to the view. They define the URL routes that are callable in your app and defines the actions that are triggered by requests.
  - **helper**: Helpers are small snippets of code that can be called in your views to help you prevent repetition - by following the `DRY` (Don't Repeat Yourself) principle.
  - **views**: Contains the templates that are filled with model data and rendered by a controller.
- **bin**: contains the installed executables files installed by bundler. They are used to prepare the environment for your app to run the exact defines executables in this bin folder. This ensure that no other version then the generates is used when your app is deployed and run in production.

- **config**: General settings for the app, including hooks (explained later) that should be performed before or after the app is loaded, setting the environment (e.g. production, development, test) and mounting other apps within the existing app under different subdomains.
  - **apps.rb**: Allows you to configure a compound app that consists of several smaller apps. Each app has it's own default route, from which requests will be handled. Here you can set site wide configurations like caching, CSRF protection, sub-app mounting, etc.
  - **boot.rb**: Basic settings for your app which will be run when you start it. Here you can turn on or off the error logging, enable internationalization and localization, load any prerequisites like HTML5 or Mailer helpers, etc.
  - **database.rb**: Define the adapters for all the environments in your application.
- **config.ru**: Contains the complete configuration options of the app, such as which port the app listens to, whenever it uses other Padrino apps as middleware and more. This file will be used when Padrino is started from the command line.
- **exe**: contains the executable script to start the app.
- **Gemfile**: The place where you declare all the necessary *gems* for your project. Bundle takes the content of this file and installs all the dependencies.
- **Gemfile.lock**: This is a file generated by Bundler after you run `bundle install` within your project. It is a listing of all the installed gems and their versions.
- **public**: Directory where you put static resources like images directory, JavaScript files, and style sheets. You can use for your asset packaging sinatra-assetpack or sprockets.

- **Rakefile**: Is the file to manage build automation tasks. For example will print the rake task `rake routes` all the defined routes in your padrino application
- **tmp**: This directory holds temporary files for intermediate processing like cache, tests, local mails, etc.

1.4 Conclusion

We have covered a lot of stuff in this chapter: installing the Padrino gem, finding the right tools to manage different Ruby versions, and creating our first Padrino app. Now it is time to jump into a real project!

Chapter 2

Job Vacancy Application

There are more IT jobs out there than there are skilled people available. It would be great if we could have the possibility to offer a platform where users can easily post new jobs vacancies to recruit people for their company. Now our job is to build this software using Padrino. We will apply **K.I.S.S** principle to obtain a simple and extensible design.

First, we are going to create the app file and folder structure. Then we are adding feature by feature until the app is complete. First, we will take a look at the basic design of our app. Afterwards, we will implement one feature at a time.

2.1 Creating The Application

Start with generating a new project with the canonical `padrino` command. In contrast to our "Hello World!" application (app) before, we are using new options:

```
$ mkdir ~/padrino-projects
$ cd ~/padrino-projects
$ padrino-gen project job-vacancy -d activerecord \
  -t rspec \
  -s jquery \
  -e erb \
  -a sqlite
```

Explanation of the fields commands:

- **-d activerecord**: We are using Active Record as the orm library (*Object Relational Mapper*).
- **-t rspec**: We are using the RSpec testing framework.
- **-s jquery**: Defining the JavaScript library we are using - for this app will be using the ubiquitous jQuery library.
- **-e erb**: We are using ERB (*embedded ruby*) markup for writing HTML templates.
- **-a sqlite**: Our adapter for the activerecord ORM[^orm] database adapter is SQLite. The whole database is saved in a text file.

Since we are using RSpec for testing, we will use its' built-in mock extensions rspec-mocks for writing tests later. In case you want to use another mocking library like rr or mocha, feel free to add it with the **-m** option.

You can use a vast array of other options when generating your new Padrino app, this table shows the currently available options:

- **`orm`**: Available options are: activerecord, couchrest, dynamoid, datamapper, minirecord, mongomapper, mongoid, mongomatic, ohm, ripple, and sequel. The command line alias is **`-d`**.
- **`test`**: Available options are: bacon, cucumber, minitest, rspec, shoulda, and test-unit. The command line alias is **`-t`**.
- **`script`**: Available options are: dojo, extcore, jquery, mootools, and prototype. The command line alias is **`-s`**.
- **`renderer`**: Available options are: erb, haml, liquid, and slim. The command line alias is **`-e`**.
- **`stylesheet`**: Available options are: compass, less, sass/scss, and scss (which ist just sass with scss syntax). The command line alias is **`-c`**.

- `mock`: Available options are: mocha and rr.

The default value of each option is none. In order to use them you have to specify the option you want to use.

Besides the `project` option for generating new Padrino apps, the following table illustrates the other generators available:

- `admin`: A built-in admin dashboard to manager your entities.
- `admin_page`: Creates for an existing model the CRUD operation for the admin interface.
- `app`: You can define other apps to be mounted in your main app.
- `controller`: A controller takes data from the models and puts them into view that are rendered.
- `mailer`: Creating new mailers within your app.
- `migration`: Migrations simplify changing the database schema.
- `model`: Models describe data objects of your application.
- `project`: Generates a completely new app from the scratch.
- `plugin`: Creating new Padrino projects based on a template file - it's like a list of commands.

Later, when *the time comes*, we will add extra gems, for now though we'll grab the current gems using `bundle`[1] by running at the command line:

```
$ bundle install
```

[1] You can also use the `-b` option during project creation - then bundle will run for your automatically.

2.1.1 Basic Layout

Lets design our first version of the *index.html* page which is the starter page our app. An early design question is: Where to put the *index.html* page? Because we are not working with controllers, the easiest thing is to put the *index.html* directly under the public folder in the project.

We are using HTML5 for the page, and add the following code into `public/index.htm`

```
<!DOCTYPE html>
<html lang="en-US">
  <head>
    <title>Start Page</title>
  </head>
  <body>
    <p>Hello, Padrino!</p>
  </body>
</html>
```

Plain static content - this used to be the way websites were created in the beginning of the web. Today, apps provide dynamic layout. During this chapter, we will how to add more and more dynamic parts to our app.

We can take a look at our new page by executing the following command:

```
$ cd job-vacancy
$ bundle exec padrino start
```

You should see a message telling you that Padrino has taken the stage, and you should be able to view our created index page by visiting http://localhost:3000/index.htm in your browser.

You might ask "Why do we use the `bundle exec` command - isn't `padrino start` enough?" The reason for this is that we use bundler to load exactly those Ruby gems that we specified in the Gemfile. I recommend that you use `bundle exec` for all following commands, but to focus on Padrino, I will skip this command on the following parts of the book.

You may have thought it a little odd that we had to manually requests the index.html in the URL when viewing our start page. This is because our app currently has no idea about **routing**. Routing is the process to recognize request

URLs and to forward these requests to actions of controllers. With other words: A router is like a like vending machine where you put in money to get a coke. In this case, the machine is the *router* which *routes* your input “Want a coke” to the action “Drop a Coke in the tray”.

2.1.2 First Controller And Routing

Lets add some basic routes for displaying our home, about, and contact-page. How can we do this? With the help of a routing controller. A controller makes data from you app (in our case job offers) available to the view (seeing the details of a job offer). Now let’s create a controller in Padrino names page:

```
$ padrino-gen controller pages --no-helper
  create  app/controllers/pages.rb
  create  app/views/pages
   apply  tests/rspec
  create  spec/app/controllers/pages_controller_spec.rb
```

Please note that we are using the `-no-helper` option which omits the creation of a helper files for our views.

Lets take a closer look at our page-controller:

```
# app/controller/pages.rb

JobVacancy::App.controllers :pages do

  # get :index, :map => '/foo/bar' do
  #   session[:foo] = 'bar'
  #   render 'index'
  # end

  # get :sample, :map => '/sample/url', :provides => [:any, :js] do
  #   case content_type
  #     when :js then ...
  #     else ...
  # end

  # get :foo, :with => :id do
  #   'Maps to url '/foo/#{params[:id]}''
  # end
```

```
  # get '/example' do
  #   'Hello world!'
  # end
end
```

The controller above defines for our `JobVacancy` the `:pages` controller with no specified routes inside the app. Let's change this and define the *about*, *contact*, and *home* actions:

```
# app/controller/pages.rb

JobVacancy::App.controllers :pages do
  get :about, :map => '/about' do
    render :erb, 'about'
  end

  get :contact , :map => '/contact' do
    render :erb, 'contact'
  end

  get :home, :map => '/' do
    render :erb, 'home'
  end
end
```

We will go through each line:

- `JobVacancy::App.controller :pages` - defines the namespace *page* for our JobVacancy app. Typically, the controller name will also be part of the route.
- `do ... end` - This expression defines a block in Ruby. Think of it as a method without a name, also called anonymous functions, which is passed to another function as an argument.
- `get :about, :map => '/about'` - The HTTP command *get* starts the declaration of the route followed by the *about* action (as a symbol[2]), and is finally mapped to the explicit URL */about*. When you

[2]Unlike strings, symbols of the same name are initialized and exist in memory only once during a session of ruby. This makes your programs more efficient.

start your server with **`bundle exec padrino s`** and visit the URL http://localhost:3000/about, you can see the rendered output of this request.

- **`render :erb, 'about'`** - This action tells us that we want to render the *erb* file *about* for the corresponding controller which is **`page`** in our case. This file is actually located at **`app/views/page/about.erb`** file. Normally the views are placed under **`app/views/<controller-name>/<action-`** Instead of using an ERB templates, you could also use **`:haml`**, or another template engine. You can even completely drop the rendering option and leave the matching completely for Padrino.

Call the following command to see all defined routes for your application:

```
$ padrino rake routes
=> Executing Rake routes ...

  Application: JobVacancy
  URL                     REQUEST  PATH
  (:pages, :about)           GET     /about
  (:pages, :contact)         GET     /contact
  (:pages, :home)            GET     /
```

2.1.3 Application Template

Although we are now able to put content (albeit static) on our site, it would be nice to have some sort of basic styling on our web page. First we need to generate a basic template for all pages we want to create:

```
<%# app/views/layouts/application.erb %>

<!DOCTYPE html>
<html lang="en-US">
  <head>
    <title>Job Vacancy - find the best jobs</title>
  </head>
  <body>
    <%= yield %>
  </body>
</html>
```

Let's see what is going on with the `<%= yield %>` line. At first you may ask what does the `<>` symbols mean. They are indicators that you want to execute Ruby code to fetch data that is put into the template. Here, the `yield` command will put the content of the called page, like *about.erb* or *contact.erb*, into the template.

2.1.4 CSS Design Using bulma

bulma is an open source CSS framework based on Flexbox. It is designed to be 100% responsive for mobile devices.

Padrino itself also provides built-in templates for common tasks done on web app. These padrino-recipes help you saving time by not reinventing the wheel. Thanks to @wikimatze, we use his bootstrap-plugin by executing:

```
$ padrino g plugin bulma
   apply  https://raw.github.com/padrino/padrino-recipes/master/ \
     plugins/bulma_plugin.rb
   create    public/stylesheets/bulma.css
```

Next we need to include the style sheet in our app template for the whole app:

```
<%# app/views/layouts/application.erb %>

<!DOCTYPE html>
<html lang="en-US">
  <head>
    <title>Job Vacancy - find the best jobs</title>
    <%= stylesheet_link_tag 'bulma' %>
    <%= javascript_include_tag 'jquery', 'jquery-ujs' %>
  </head>
  <body>
    <%= yield %>
  </body>
</html>
```

The `stylesheet_link_tag` points to the *bootstrap.min.css* in you app *public/stylesheets* directory and will automatically create a link to this stylesheet. You can also use `javascript_include_tag` which does the same as `stylesheet_link` just for JavaScript files.

2.1.5 Using Sprockets to Manage the Asset Pipeline

Sprockets is a way to manage serving your assets like CSS, and JavaScript compiling all the different files in one summarized file for each type.

To implement Sprockets in Padrino there the following strategies:

- rake-pipeline: Define filters that transforms directory trees.
- grunt: Set a task to compile and manage assets in JavaScript.
- sinatra-asset-pipeline: Let's you define you assets transparently in Sinatra.
- sprocket-helpers: Asset path helpers for Sprockets 2.0 applications
- padrino-sprockets: Integrate sprockets with Padrino in the Rails way.

We are using the **padrino-sprockets** gem. Let's add it to our Gemfile (don't forget to run `bundle install`):

```
# Gemfile

gem 'padrino-sprockets', :require => ['padrino/sprockets'],
  :git => 'git://github.com/nightsailer/padrino-sprockets.git'
```

Next we need to move all our assets from the public folder in the assets folder:

```
$ mkdir -p job-vacancy/app/assets
$ cd job-vacancy/public
$ mv images ../app/assets
$ mv javascripts ../app/assets
$ mv stylesheets ../app/assets
```

Now we have to register Padrino-Sprockets in this application:

```ruby
# app/app.rb

module JobVacancy
  class App < Padrino::Application
    ...
    register Padrino::Sprockets
    sprockets
    ...
  end
end
```

Next we need create an application.css file and add the following to determine the order of the loaded CSS files in **`app/assets/stylesheets/application.css`**

```css
/* app/assets/stylesheets/application.css */

/*
 * This is a manifest file that'll automatically include all the stylesheets ...
 * ...
 *
 *= require_self
 *= require bulma
 *= require site
*/
```

This file serves as a manifest file and the **`require_self`** directive indicates that any CSS in the file should be delivered in the given order to the browser.

First we are loading the **`bulma`** css, and then our customized **`site`** CSS. This is helpful if you want to check the order of the loaded CSS as a comment above your application without ever have to look into the source of it. The file

Next let's have a look into our JavaScript file **`app/assets/javascript/applicatio`**

```javascript
/* app/assets/javascript/application.js */

// This is a manifest file that'll be compiled into including all the files ...
// ...
//
//= require_tree .
```

The interesting thing here is the **`require_tree .`** option. This option (note the Unix dot operator) tells Sprockets to include all JavaScript files in the