Sessions

Introduction

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.

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.

Why Padrino With The Developer Point of View

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 so small it is easy to go through the code to understand most of the source. There is no need for monkey-patching, almost everything can be changed via an API. Padrino is rack-friendly, so 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.

Why Padrino In A Human Way?

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. The ones available are: datamapper, sequel, activerecord, mongomapper, mongoid, and couchrest.
- `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**"

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 basically 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
- Aptana RadRails - 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.

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 the nebulous programmer why the lucky stiff in a entertaining and educational way.

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

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 &#39;export PATH="$HOME/.rbenv/bin:$PATH"&#39; >> ~/.bashrc
```

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

```
$ echo &#39;eval "$(rbenv init -)"&#39; >> ~/.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 1.9.3-p392`:

```
$ rbenv install 1.9.3-p392
```

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 1.9.3-p392 the current executable on your machine:

```
$ rbenv global 1.9.3-p392
```

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

```
$ 1.9.3-p392 (set by /home/.rbenv/versions)
```

Now you are a ready to hack on with Padrino!

%%/* vim: set ts=2 sw=2 textwidth=120: */

Hello world

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

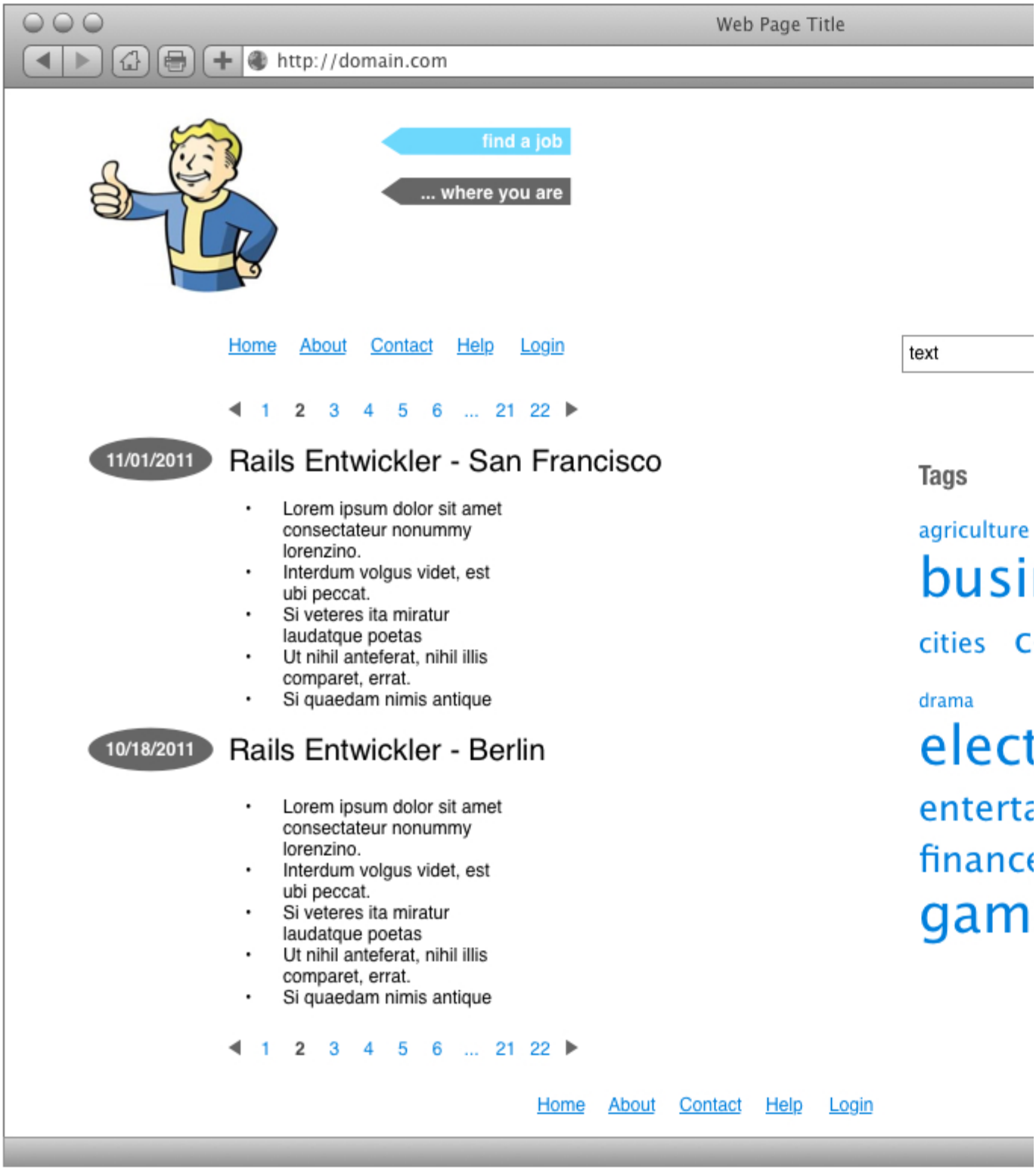

Figure 1-1. Start page of the app

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 **0.11.2**). 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-world
```

Let's go through each part of this command:

- `padrino generate` : 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-world`: 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  tmp
    create  .components
    create  app
    create  app/app.rb
```

```
      create  app/controllers
      create  app/helpers
      create  app/views
      create  app/views/layouts
      create  Gemfile
      create  Rakefile
    skipping  orm component...
    skipping  test component...
    skipping  mock component...
    skipping  script component...
    applying  slim (renderer)...
       apply  renderers/slim
      insert  Gemfile
    skipping  stylesheet component...
   identical  .components
       force  .components
       force  .components

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

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. This makes it very easy for other developers 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, execute the following command and check the console output:

```
    $ gem install bundler
        Fetching: bundler-1.3.5.gem (100%)
```

```
      Successfully installed bundler-1.3.5
      1 gem installed
```

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

```
$ cd hello-world
$ bundle
  Fetching gem metadata from http://rubygems.org/.........

  Using rake ...
  Using ...
  Your bundle is complete! Use `bundle show [gemname]` to see where a bundled gem
```

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`:

```
module HelloWorld
  class App < Padrino::Application

    get "/" do
      "Hello World!"
    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.

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-world* project. The app consists of the following parts:

```
|-- Gemfile
|-- Gemfile.lock
|-- app
|   |-- app.rb
|   |-- controllers
|   |-- helpers
|   `-- views
|       `-- layouts
|-- config
|   |-- apps.rb
|   |-- boot.rb
|   `-- database.rb
|-- config.ru
|-- public
|   |-- favicon.ico
|   |-- images
|   |-- javascripts
|   `-- stylesheets
`-- tmp
```

We will go through each part.

- **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 ins` within your project. It is a listing of all the installed gems and their versions.
- **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.
- **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 configs 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 prerquisites 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.
- **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.
- **tmp**: This directory holds temporary files for intermediate processing like cache, tests, local mails, etc.

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!

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 very easy 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.

Creating a new 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 g project job-vacancy -d activerecord -t rspec -s jquery -e erb -a sqli
```

Explanation of the fields commands:

- **g**: Is shortcut for `generate`.
- **-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. An alternative is Haml or Slim, but to keep the project as simple as possible, we stick with ERB. Feel free to use them if you like to.

- **-a sqlite**: Our ORM[^orm] database adapter is SQLite. It is easy to install because 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`, `datamapper`, `mongoid`, `mongom` , and `sequel`. The command line alias is `-d`.
- `test`: Availabe options are: `bacon`, `cucumber`, `minitest`, `riot`, `rspec`, `should` , and `testspec`. The command line alias is `-t`.
- `script`: Available options are: `dojo`, `extcore`, `jquery`, `mootools`, `prototype` , and `rightjs`. 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` , and `scss`. The command line alias is `-c`.
- `mock`: Available options are: `mocha` and `rr`.

The default value of each option is none. So to 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 very nice built-in admin dashboard.
- `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 date 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` by running at the command line:

```bash $ bundle install ```

Basic Layout Template

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/ind`
:

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

```
    </body>
  </html>
```

Explanation of the parts:

- `<!DOCTYPE html>` - The *document type* tells the browser which HTML version should be used for rendering the content correctly.
- `<head>...</head>` - Specifying meta information like title, description, and other things, this is also the place to where to add CSS and JavaScript files.
- `<body>...</body>` - In this section the main content of the page is displayed.

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 se 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.html in your browser.

But hey, you might ask "Why do we use the `bundle exec` command - isn't just `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 requeste 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".

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 g controller page
```

The output of this command is:

```
create  app/controllers/page.rb
create  app/helpers/page_helper.rb
create  app/views/page
 apply  tests/rspec
create  spec/app/controllers/page_controller_spec.rb
```

(If you have questions about the output above, please drop me a line - I think it is so clear that it doesn't need any explanation about it.)

Lets take a closer look at our page-controller:

```
# app/controller/page.rb

JobVacancy::App.controllers :page do
```

```
    # get :index, :map => &#39;/foo/bar&#39; do
    #   session[:foo] = &#39;bar&#39;
    #   render &#39;index&#39;
    # end

    # get :sample, :map => &#39;/sample/url&#39;, :provides => [:any, :js] do
    #   case content_type
    #     when :js then ...
    #     else ...
    # end

    # get :foo, :with => :id do
    #   &#39;Maps to url &#39;/foo/#{params[:id]}&#39;&#39;
    # end

    # get &#39;/example&#39; do
    #   &#39;Hello world!&#39;
    # end

  end
```

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

```
  # app/controller/page.rb

  JobVacancy:.App.controllers :page do
    get :about, :map => &#39;/about&#39; do
      render :erb, &#39;page/about&#39;
    end

    get :contact , :map => &#39;/contact&#39; do
      render :erb, &#39;page/contact&#39;
    end

    get :home, :map => &#39;/&#39; do
      render :erb, &#39;page/home&#39;
    end

  end
```

We will go through each line:

- `JobVacancy::App.controller :page` - Define 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 (in the form of a Ruby symbol[1]), and is finally mapped to the explicit URL */about*. When you 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, 'page/about'` - This action tells us that we want to render an the *erb* file *page/about*. This file is actually located at `app/views/page/about.e` file. Normally the views are placed under *app/views//*. Instead of using an ERB templates, you could also use `:haml`, or another template language. If you are lazy, you can leave the option for the rendering option completely out and leave the matching completely for Padrino.

To see what routes you have defined for your app just call `padrino rake routes` :

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

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

This command crawls through your app looking for delicious routes and gives you a nice overview about **URL, REQUEST**, and **PATH**.

App Template With ERB

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. Lets 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.

CSS design using Twitter bootstrap

The guys at Twitter were kind enough to make their CSS framework **Twitter Bootstrap** available for everyone to use. It is available from Github at public repository on Github.

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 [@arthur_chiu] (http://twitter.com/#!/arthur_chiu "@arthur_chiu"), we use his bootstrap-plugin by executing:

```
$ padrino-gen plugin bootstrap

  apply  https://github.com/padrino/padrino-recipes/raw/master/plugins/bootstrap_
  create    public/stylesheets/bootstrap.css
  create    public/stylesheets/bootstrap-responsive.css
  create    public/javascripts/bootstrap.js
  create    public/javascripts/bootstrap.min.js
  create    public/images/glyphicons-halflings.png
  create    public/images/glyphicons-halflings-white.png
```

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

```
<!DOCTYPE html>
<html lang="en-US">
  <head>
    <title>Job Vacancy - find the best jobs</title>
    <%= stylesheet_link_tag &#39;bootstrap&#39;, &#39;bootstrap-responsive&#39; %
    <%= javascript_include_tag &#39;bootstrap.min&#39;, &#39;jquery&#39;, &#39;jq
  </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. The `javascript_include_tag` does the same as `stylesheet_link_tag` for your JavaScript files in the *public/javascripts* directory.

Using Sprockets to Manage the Asset Pipeline

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

each type. They make it easy to take advantage to use a preprocessor to write your assests with Sass, Coffesscript, or LESS.

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-assetpack: Let's you define you assets transparently in Sinatra.
- padrino-sprockets: Integrate sprockets with Padrino in the Rails way.

We are using the **padrino-sprockets** gem. Let's add it to our Gemfile:

```
# Gemfile
gem &#39;padrino-sprockets&#39;, :require => [&#39;padrino/sprockets&#39;], :git
```

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

```
$ cd <path-to-your-padrino-app>
$ mkdir -p app/assets
$ mv public/javascript app/assets
$ mv public/stylesheets app/assets
$ mv public/images app/assets
```

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

```
# app/app.rb
module JobVacancy
  class App < Padrino::Application
    ...
    register Padrino::Sprockets
    sprockets
    ...
  end
end
```

Next we need to determine the order of the loaded CSS files:

```
# app/assets/stylesheets/application.css
/*
 * This is a manifest file that&#39;ll automatically include all the stylesheets
 * and any sub-directories. You&#39;re free to add application-wide styles to thi
 * the top of the compiled file, but it&#39;s generally better to create a new fi
 * require_self: Puts the CSS contained within this file at the precise location
 * at the top of the generated css file
 * require_tree . means, that requiring all stylesheets from the current director
 *
 *= require_self
 *= require bootstrap
 *= require bootstrap-responsive
 *= require site
*/
```

First we are loading the `bootstrap` default css, then `bootstrap-response`, and finally our customized `site` CSS. The `require_self` loads the file itself, to define the order that the files are loaded. 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.

Next let's have a look into our JavaScript files:

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

// This is a manifest file that&#39;ll be compiled into including all the files l
// Add new JavaScript/Coffee code in separate files in this directory and they&#3
// be included in the compiled file accessible from http://example.com/assets/app
// It&#39;s not advisable to add code directly here, but if you do, it&#39;ll app
// the compiled file.
//
//= require_tree .
```

The interesting thing here is the `require_tree .` option. This option tells Sprockets to include all JavaScript files in the assets folder with no specific order.

Now, we can clean up the include statements in our application

template:

```
# app/views/application.erb

<!DOCTYPE html>
<html lang="en-US">
<head>
  <title>Job Vacancy - find the best jobs</title>
  <%= stylesheet_link_tag &#39;/assets/application&#39; %>
  <%= javascript_include_tag &#39;/assets/application&#39; %>
</head>
```

Now we want to enable compression for our CSS and JavaScript files. For CSS compression Padrino Sprockets is using YUI compressor and for JS compression the Uglifier. We need to add these these Gems in our `Gemfiles`:

```
# Gemfile
...
gem &#39;padrino-sprockets&#39;, :require => &#39;padrino/sprockets&#39;, :git =>
gem &#39;uglifier&#39;, &#39;2.1.1&#39;
gem &#39;yui-compressor&#39;, &#39;0.9.6&#39;
```

And finally we need to enable minifying in our production environment:

```ruby # app/app.rb module JobVacancy class App < Padrino::Application ... register Padrino::Sprockets sprockets :minify => (Padrino.env == :production) end end

## *Navigation*

Next we want to create the top-navigation for our app. So we already implemented the *page* controller with the relevant actions. All we
```

need is to put links to them in a navigation header for our basic layout.

```
<!DOCTYPE html>
<html lang="en-US">
  <head>
    <title>Job Vacancy - find the best jobs</title>
    <%= stylesheet_link_tag &#39;bootstrap&#39;, &#39;bootstrap-responsive&#39; %
    <%= javascript_include_tag &#39;bootstrap.min&#39;, &#39;jquery&#39;, &#39;jq
    <%= stylesheet_link_tag &#39;/stylesheets/site.css&#39; %>
</head>
<body>
  <div class=="container">
    <div class="row">
      <div class="span12 offset3">
        <span id="header">Job Vacancy Board</span>
      </div>
      <div class="row">
        <nav id="navigation">
          <div class="span2 offset4">
            <%= link_to &#39;Home&#39;, url_for(:page, :home) %>
          </div>
          <div class="span2">
            <%= link_to &#39;About&#39;, url_for(:page, :about) %>
          </div>
          <div class="span2">
            <%= link_to &#39;Contact&#39;, url_for(:page, :contact) %>
          </div>
        </nav>
      </div>
      <div class="row">
        <div class="span9 offset3 site">
          <%= yield %>
        </div>
      </div>
    </div>
  </div>
</body>
```

Explanation of the new parts:

- `link_to` - Is a helper for creating links. The first argument is the name for the link and the second is for the URL (href) to which the

link points to.

- `url_for` - This helper return the link which can be used as the second parameter for the `link_to` function. It specifies the `<:controller>, <:a` which will be executed. You can use in your helper in your whole app to create clean and encapsulated URLs.

Now that the we provide links to other parts of the app, lets add some sugar-candy styling:

```
# app/assets/stylesheets/site.css

body {
  font: 18.5px Palatino, &#39;Palatino Linotype&#39;, Helvetica, Arial, Verdana,
  text-align: justify;
}

#header {
  font-family: Lato;
  font-size: 40px;
  font-weight: bold;
}

#navigation {
  padding-top: 20px;
}

h1 {
  font-family: Lato;
  font-size: 30px;
  margin-bottom: 20px;
}

.site {
  padding: 20px;
  line-height: 1.8em;
}
```

I will not explain anything at this point about CSS. If you still don't know how to use it, please go through w3c school css tutorial. Since we are using the asset pipeline, we don't need to register our new CSS file in `views/application.erb` - now you will understand why we did this.

Writing Tests

Our site does not list static entries of job offers that you write, but other users will be allowed to post job offers from the internet to our site. We need to add this behavior to our site. To be on the sure side, we will implement this behavior by writing tests first, then the code. We use the RSpec testing framework for this.

Remember when we created the *page-controller* with `padrino g controller page` ? Thereby, Padrino created a corresponding spec file *spec/app/controller/page_controller_spec.rb* which has the following content:

```
require &#39;spec_helper&#39;

describe "PageController" do
  before do
    get "/"
  end

  it "returns hello world" do
    last_response.body.should == "Hello World"
  end
end
```

Let's update that file and write some basic tests to make sure that everything is working as expected. Replace the specs in the file with the following code:

```
require &#39;spec_helper&#39;

describe "PageController" do

  describe "GET #about" do

    it "renders the :about view" do
      get &#39;/about&#39;
```

```
        last_response.should be_ok
      end
    end

    describe "GET #contact" do

      it "renders the :contact view" do
        get &#39;/contact&#39;
        last_response.should be_ok
      end
    end

    describe "GET #home" do
      it "renders :home view" do
        get &#39;/&#39;
        last_response.should be_ok
      end
    end

  end
```

Let's explain the interesting parts:

- `spec_helper` - Is a file to load commonly used functions to setup the tests.
- `describe block` - This block describes the context for our tests. Think of it as way to group related tests.
- `get ...` - This command executes a HTTP GET to the provided address.
- `last_response` - The response object returns the header and body of the HTTP request.

Now let's run the tests with `rspec spec/app/controllers/page_controller_spec.rb` and see what's going on:

```
...

Finished in 0.21769 seconds
3 examples, 0 failures
```

Cool, all tests passed! We didn't exactly use behavior-driven

development until now, but will do so in the next parts.

Note: It's possible your tests did not pass due to a Padrino error in which a comma (,) was ommited during the initial app generation that looks something like 'NameError: undefined local variable' so check your `spec_helper.rb` file and make sure the following matches:

```
def app(app = nil, &blk) # note the comma right after nil
```

Note: It's possible your tests did not pass due to a Padrino error in which a comma (,) was ommited during the initial app generation that looks something like 'NameError: undefined local variable' so check your `spec_helper.rb` file and make sure the following matches:

```
def app(app = nil, &blk) # note the comma right after nil
```

I> ## Red-Green Cycle I> I> In behavior-driven development (BDD) it is important to write a failing test first and then the code that satisfies the I> test. The red-green cycle represents the colors that you will see when executing these test: Red first, and then I> beautiful green. But once your code passes the tests, take yet a little more time to refactor your code. This little I> mind shift helps you a lot to think more about the problem and how to solve it. The test suite is a nice by product too.

Creation Of The Models

User Model

There are many different ways how to develop a user entity for your system. A user in our system will have an *unique* identification number **id**, a **name**, and an **email**. We can specify the location of

the model by appending the end of the generate command with `-a app` as follows:

```
$ padrino g model user name:string email:string -a app

   apply  orms/activerecord
   apply  tests/rspec
  create  app/models/user.rb
  create  spec/app/models/user_spec.rb
  create  db/migrate/001_create_users.rb
```

Wow, it created a quite a bunch of files for us. Let's examine each of them:

user.rb

```
# app/models/user.rb

class User < ActiveRecord::Base
end
```

All we have is an empty class which inherits from ActiveRecord::Base . `ActvieRecord` provides a simple object-relational-mapper from our models to corresponding database tables. You can also define relations between models through associations.

```
# spec/app/models/user_spec.rb

require &#39;spec_helper&#39;

describe "User Model" do
  let(:user) { User.new }
  it &#39;can be created&#39; do
    user.should_not be_nil
  end
end
```

As you can see, the generator created already a test for us, which basically checks if the model can be created. What would happen if you run the tests for this model? Let the code speak of it's own and

run the tests, that's what they are made for after all:

```
$ rspec spec/app/models

User Model
  can be created (FAILED - 1)

Failures:

  1) User Model can be created
     Failure/Error: let(:user) { User.new }
     ActiveRecord::StatementInvalid:
       Could not find table &#39;users&#39;
     # ./spec/app/models/user_spec.rb:4:in `new&#39;
     # ./spec/app/models/user_spec.rb:4:in `block (2 levels) in <top (required)>&
     # ./spec/app/models/user_spec.rb:6:in `block (2 levels) in <top (required)>&

Finished in 0.041 seconds
1 example, 1 failure

Failed examples:
```

Executing the test resulted in an error. However, it very explicitly told us the reason: The *user* table does not exist yet. And how do we create one? Here, migrations enter the stage.

Migrations helps you to change the database in an ordered manner. Let's have a look at our first migration:

```
db/migrate/001_create_users.rb

class CreateUsers < ActiveRecord::Migration
  def self.up
    create_table :users do |t|
      t.string :name
      t.string :email
      t.timestamps
    end
  end

  def self.down
```

```
      drop_table :users
    end
  end
```

This code will create a `users` table with the `name` and `email` attributes. The `id` attribute will be created automatically unless you specify to use a different attribute as the unique key to a database entry. By the way, the convention to name tables of models in the plural form comes from Ruby On Rails. Now we need to run this migration:

```
$ padrino rake ar:migrate

=> Executing Rake ar:migrate ...
  DEBUG -  (0.1ms)  select sqlite_version(*)
  DEBUG -  (143.0ms)  CREATE TABLE "schema_migrations" ("version" varchar(255) NC
  DEBUG -  (125.2ms)  CREATE UNIQUE INDEX "unique_schema_migrations" ON "schema_m
  DEBUG -  (0.2ms)  SELECT "schema_migrations"."version" FROM "schema_migrations"
   INFO - Migrating to CreateUsers (1)
  DEBUG -  (0.1ms)  begin transaction
  ...
```

Since we are working in the development environment, Padrino automatically created the development database for us:

```
$ ls db/
  job_vacancy_development.db  job_vacancy_test.db  migrate  schema.rb
```

Now let's start sqlite3, connect to the database, and see if the users table was created properly:

```
$ sqlite3 db/job_vacancy_development.db

SQLite version 3.7.13 2012-06-11 02:05:22
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> .tables
schema_migrations  users
sqlite> .schema users
CREATE TABLE "users" ("id" INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL, "name" var
sqlite> .exit
```

Let's have a look on the `config/database.rb` file to understand more about the different databases:

```
ActiveRecord::Base.configurations[:development] = {
  :adapter => &#39;sqlite3&#39;,
  :database => Padrino.root(&#39;db&#39;, &#39;job_vacancy_development.db&#39;)
}

ActiveRecord::Base.configurations[:production] = {
  :adapter => &#39;sqlite3&#39;,
  :database => Padrino.root(&#39;db&#39;, &#39;job_vacancy_production.db&#39;)
}

ActiveRecord::Base.configurations[:test] = {
  :adapter => &#39;sqlite3&#39;,
  :database => Padrino.root(&#39;db&#39;, &#39;job_vacancy_test.db&#39;)
}
```

As you can see, each of the different environments *development*, *production*, and *test* have their own database. Lets's be sure that all databases are created:

```
$ padrino rake ar:create:all

bundle exec padrino rake ar:create:all
=> Executing Rake ar:create:all ...
/home/elex/Dropbox/git-repositorie/job-vacancy/db/job_vacancy_development.db alre
/home/helex/Dropbox/git-repositorie/job-vacancy/db/job_vacancy_development.db alr
/home/helex/Dropbox/git-repositories/job-vacancy/db/job_vacancy_production.db alr
/home/helex/Dropbox/git-repositories/job-vacancy/db/job_vacancy_test.db already e
/home/helex/Dropbox/git-repositories/job-vacancy/db/job_vacancy_test.db already e
```

Alright, now we are ready to re-execute the tests again.

```
$ rspec spec/app/models

User Model
  can be created (FAILED - 1)

Failures:

  1) User Model can be created
     Failure/Error: let(:user) { User.new }
```

```
        ActiveRecord::StatementInvalid:
          Could not find table &#39;users&#39;
        # ./spec/app/models/user_spec.rb:4:in `new&#39;
        # ./spec/app/models/user_spec.rb:4:in `block (2 levels) in <top (required)>&
        # ./spec/app/models/user_spec.rb:6:in `block (2 levels) in <top (required)>&

    Finished in 0.04847 seconds
    1 example, 1 failure

    Failed examples:

    rspec ./spec/app/models/user_spec.rb:5 # User Model can be created
```

But why are the tests still failing? Because the migration for the *user* table was not executed for the test environment. Let's fix this with the following command:

```
    $ padrino rake ar:migrate -e test
    => Executing Rake ar:migrate ...
    ==  CreateUsers: migrating ==================================================
    -- create_table(:users)
       -> 0.0030s
    ==  CreateUsers: migrated (0.0032s) =========================================
```

Finally the test passes:

```
    $ rspec spec/app/models

    User Model
      can be created

    Finished in 0.05492 seconds
    1 example, 0 failures
```

How can we run all the tests in our application and see if everything is working? Just execute `padrino rake spec` to run all tests in the `spec/` folder:

```
    $ padrino rake spec
    => Executing Rake spec ...
    /home/helex/.rbenv/versions/1.9.3-p392/bin/ruby -S rspec ./spec/app/models/user_s
```

```
User Model
  can be created

Finished in 0.05589 seconds
1 example, 0 failures
/home/helex/.rbenv/versions/1.9.3-p392/bin/ruby -S rspec ./spec/app/controllers/p

PageController
  GET #about
    renders the :about view
  GET #contact
    renders the :contact view
  GET #home
    renders the :home view

Finished in 0.20325 seconds
3 examples, 0 failures
```

This is very handy to make sure that you didn't break anything in the existing codebase when you are working on a new feature. Run these regression tests frequently and enjoy it to see your app growing feature by feature.

Job Offer Model

Since we now know how to create the basic model of our users, it's time to create a model for presenting a job offer. A job offer consists of the following attributes:

- title: The name of the job position.
- location: The geographical location of the job.
- description: Details about the position.
- contact: An email address of the contact person.
- time-start: The earliest entry date for this position.
- time-end: A job offer isn't valid forever.

Let's run the Padrino command to create the model for us. As you see,

we once again run `-a app` at the end of our generation. Without specifying location, a new folder called `models` is created in the main directory.

```
$ padrino g model job_offer title:string location:string description:text contact
   apply  orms/activerecord
   apply  tests/rspec
   create  app/models/job_offer.rb
   create  spec/app/models/job_offer_spec.rb
   create  db/migrate/002_create_job_offers.rb
```

Perhaps your forgot to add `-a app` to specify the app directory when running the generation. You can delete a model in Padrino by running:

```
padrino g model job_offer -d
```

Keep in mind that you will need to manually delete the now empty `models` folder or with the `$ rmdir models` unix/linux command.

Next, we need to run our new database migration so that our database has the right scheme:

```
bundle exec padrino rake ar:migrate
=> Executing Rake ar:migrate ...
  DEBUG -  (0.4ms)  SELECT "schema_migrations"."version" FROM "schema_migrations"
   INFO - Migrating to CreateUsers (1)
   INFO - Migrating to CreateJobOffers (2)
  DEBUG -  (0.3ms)  select sqlite_version(*)
  DEBUG -  (0.2ms)  begin transaction
==  CreateJobOffers: migrating ================================================
-- create_table(:job_offers)
...
```

In order to run our tests, we also need to run our migrations for the test environment:

```
$ padrino rake ar:migrate -e test
=> Executing Rake ar:migrate ...
==  CreateJobOffers: migrating ================================================
```

```
   -- create_table(:job_offers)
      -> 0.0302s
   ==  CreateJobOffers: migrated (0.0316s) ======================================
```

TBD: Find a way to run ar:migrate for all environments (mainly production and test)

If you run your tests with `padrino rake spec`, everything should be fine.

Creating Connection Between User And Job Offer Model

Since we now have created our two main models, it's time to define associations between them. Associations make common operations like deleting or updating data in our relational database easier. Just imagine that we have a user in our app that added many job offers in our system. Now this customer decides that he wants to cancel his account. We decide that all his job offers should also disappear in the system. One solution would be to delete the user, remember his id, and delete all job offers entries that originate from this id. This manual effort disappears when associations are used: It becomes as easy as "If I delete this user from the system, delete automatically all corresponding jobs for this user".

We will quickly browse through the associations.

has_many

This association is the most commonly used one. It does exactly as it tells us: One object has many other objects. We define the association between the user and the job offers as shown in the following expression:

```
   # app/models/user.rb
```

```
class User < ActiveRecord::Base
  has_many :job_offers
end
```

belongs_to

The receiving object of the *has_many* relationship defines that it belongs to exactly one object, and therefore:

```
# app/models/job_offer.rb

class JobOffer < ActiveRecord::Base
  belongs_to :user
end
```

Migrate after associate

Whenever you modify your models, remember that you need to run migrations too. Because we added the associations manually, we also need to write the migrations. Luckily, Padrino helps us with this task a bit. We know that the job offer is linked to a user via the user's id. This foreign key relationship results in adding an extra column `user_id` to the `job_offers table` . For this change, we can use the following command to create a migration:

```
$ padrino g migration AddUserIdToJobOffers user_id:integer
apply  orms/activerecord
create  db/migrate/003_add_user_id_to_job_offers.rb
```

Let's take a look at the created migration:

```
class AddUserIdToJobOffers < ActiveRecord::Migration
  def self.up
    change_table :joboffers do |t|
      t.integer :user_id
    end
  end

  def self.down
    change_table :joboffers do |t|
```

```
        t.remove :user_id
      end
    end
  end
```

Can you see the small bug? This migration won't work, you have to change `joboffers` to `job_offers`. For the time being, generators can help you to write code, but not prevent you from thinking.

Finally let's run our migrations:

```
$ padrino rake ar:migrate
$ padrino rake ar:migrate -e test
```

Testing our associations in the console

To see whether the migrations were executed, we connected to the sqlite3 database via the command line. Let's use a different approach and use the Padrino console this time. All you have to do is to run the following command:

```
$ padrino c
=> Loading development console (Padrino v.0.11.1)
=> Loading Application JobVacancy
>>
```

Now you are in an environment which acts like IRB, the *Interactive Ruby* shell. This allows you to execute Ruby commands and immediately see it's response.

Let's run the shell to create a user with job offers:

```
user = User.new(:name => &#39;Matthias Günther&#39;, :email => &#39;matthias.guen
=> #<User id: nil, name: "Matthias Günther", email: "matthias.guenther", created_
>> user.name
=> "Matthias Günther"
```

This creates a user object in our session. If we want to add an entry permanently into the database, you have to use *create* method:

```
User.create(:name => &#39;Matthias Günther&#39;, :email => &#39;matthias.guenther
DEBUG -  (0.2ms)  begin transaction
  DEBUG - SQL (114.6ms)  INSERT INTO "users" ("created_at", "email", "name", "upd
  [["created_at", 2012-12-26 08:32:51 +0100], ["email", "matthias.guenther"], ["n
  ["updated_at", 2012-12-26 08:32:51 +0100]]
    DEBUG -  (342.0ms)  commit transaction
=> #<User id: 1, name: "Matthias Günther", email: "matthias.guenther", created_at
    updated_at: "2012-12-26 08:32:51">
  >>
```

Please note that now you have an entry in your development database `db/job_vacancy_development.db` . To see this, connect to the database and execute a 'SELECT' statement::

```
$ sqlite3 db/job_vacancy_development.db
SQLite version 3.7.13 2012-06-11 02:05:22
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> SELECT * FROM users;
1|Matthias Günther|matthias.guenther|2012-12-26 08:32:51.323349|2012-12-26 08:32:
sqlite>.exit
```

Since we have an user, it's time to some job offers too:

```
 $ padrino c
 => Loading development console (Padrino v.0.10.7)
 => Loading Application JobVacancy
  JobOffer.create(:title => &#39;Padrino Engineer&#39;,
    :location => &#39;Berlin&#39;,
    :description => &#39;Come to this great place&#39;,
    :contact => &#39;recruter@padrino-company.org&#39;,
    :time_start => &#39;2013/01/01&#39;,
    :time_end => &#39;2013/03/01&#39;,
    :user_id => 1)
   ...
    => #<JobOffer id: 1, title: "Padrino Engineer", location: "Berlin", descripti
    contact: "recruter@padrino-firm.org", time_start: "2013-01-01", time_end: "20
    10:12:07", updated_at: "2012-12-26 10:12:07", user_id: 1>
```

And now let's create a second one for our first user:

```
 >> JobOffer.create(:title => &#39;Padrino Engineer 2&#39;,
```

```
        :location => &#39;Berlin&#39;,
        :description => &#39;Come to this great place&#39;,
        :contact => &#39;recruter@padrino-company.org&#39;,
        :time_start => &#39;2013/01/01&#39;,
        :time_end => &#39;2013/03/01&#39;,
        :user_id => 1)
      ...
        => #<JobOffer id: 2, title: "Padrino Engineer 2", location: "Berlin", descri
        contact: "recruter@padrino-firm.org", time_start: "2013-01-01", time_end: "2
        10:41:29", updated_at: "2012-12-26 10:41:29", user_id: 1>
```

Now it's time to test our association between the user and the job-offer model. We will use the `find_by_id` method to get the user from our database, and the `job_offers` method to get all the job-offers from the user.

```
>> user = User.find_by_id(1)
  DEBUG - User Load (0.6ms)  SELECT "users".* FROM "users" WHERE "users"."id" = 1
  => #<User id: 1, name: "Matthias Günther", email: "matthias.guenther", created_
  "2012-12-26 08:32:51">
>> user.job_offers
  DEBUG - JobOffer Load (0.6ms)  SELECT "job_offers".* FROM "job_offers" WHERE "j
  => [#<JobOffer id: 1, title: "Padrino Engineer", location: "Berlin", descriptic
  contact: "recruter@padrino-firm.org", time_start: "2013-01-01", time_end: "2013
  10:12:07", updated_at: "2012-12-26 10:12:07", user_id: 1>, #<JobOffer id: 2, ti
  "Berlin", description: "Come to this great place", contact: "recruter@padrino-f
  time_end: "2013-03-01", created_at: "2012-12-26 10:41:29", updated_at: "2012-12
```

Here you can see the advantage of using associations: When you declare them, you automatically get methods for accessing the data you want.

Ok, we are doing great so far. With users and job offers in place, let's add some tests to create and associate these objects.

Testing our app with RSpec + Factory Girl

When you use data for the tests, you need to decide how to create them. You could, of course, define a set of test data with pure SQL and add it to your app. A more convenient solution instead is to use

factories and fixtures. Think of factories as producers for you data. You are telling the factory that you need 10 users that should have different names and emails. This kind of mass object creation which are called fixtures in testing, can easily be done with Factory Girl. Factory Girl defines it's own language to create fixtures in an `ActiveRecord` -like way, but with a much cleaner syntax.

What do we need to use Factory Girl in our app? Right, we first we need to add a gem to our `Gemfile`:

```
# Gemfile
...
gem &#39;factory_girl&#39;, &#39;4.2.0&#39;, :group => &#39;test&#39;
```

If you pay a closer look into the `Gemfile`, you can see that we have several gems with the `:group` option:

```
# Gemfile
...
gem &#39;rspec&#39; , &#39;2.13.0&#39;, :group => &#39;test&#39;
gem &#39;factory_girl&#39;, &#39;4.2.0&#39;, :group => &#39;test&#39;
gem &#39;rack-test&#39;, &#39;0.6.2&#39;, :require => &#39;rack/test&#39;, :group
```

Luckily we can use the `:group <name> do ... end` syntax to cleanup to get rid of several `:group => 'test'` lines in our `Gemfile`:

```
# Gemfile

group :test do
  gem &#39;rspec&#39; , &#39;2.13.0&#39;
  gem &#39;factory_girl&#39;, &#39;4.2.0&#39;
  gem &#39;rack-test&#39;, &#39;0.6.2&#39;, :require => &#39;rack/test&#39;
end
```

Execute `bundle` and the new gem will be installed.

Next we need to define a *factory* to include all the fixtures of our models:

```
# spec/factories.rb

# encoding: utf-8
FactoryGirl.define do

  factory :user do
    name "Matthias Günther"
    email "matthias.guenther@wikimatze.de"
  end

end
```

I want to add myself as a test user. Since I'm German, I want to use special symbols, called umlauts, from my language. To make Ruby aware of this, I'm putting `# encoding: utf-8` at the header of the file. The symbol `:user` stands for the definition for user model. To make our factory available in all our tests, we just have to *require* our factory in the `spec_helper.rb`:

```
# spec/spec_helper.rb

PADRINO_ENV = &#39;test&#39; unless defined?(PADRINO_ENV)
require File.expand_path(File.dirname(__FILE__) + "/../config/boot")
require File.dirname(__FILE__) + "/factories"
...
```

Now we have everything at hand to create a user with the factory while testing our app:

```
# spec/app/models/user_spec.rb

require &#39;spec_helper&#39;

describe "User Model" do
  let(:user) { FactoryGirl.build(:user) }
  let(:job_offer) { {:title => &#39;Padrino Engineer&#39;, :location => &#39;Berl
  it &#39;can be created&#39; do
    user.should_not be_nil
  end

  it &#39;fresh user should have no offers&#39; do
    user.job_offers.size.should == 0
```

```
    end

    it &#39;have job-offers&#39; do
      user.job_offers.build(job_offer)
      user.job_offers.size.should == 1
    end

  end
```

The basic philosophy behind testing with fixtures is that you create objects as you need them with convenient expressions. Instead of using `User.create`, we are using `FactoryGirl.build(:user)` to temporarily create a `user` fixture. The job offer that we are adding for the tests is defined as an attribute hash - you map the attributes (keys) to their values. If you run the tests, they will pass.

The `build` method that we use to create the user will only add the test object in memory. If you want to permanently add fixtures to the database, you have to use `create` instead. Play with it, and see that the same test using `create` instead of `build` takes much longer because it hits the database.

We can improve our test by creating a factory for our job offer too and cleaning the `user_spec.rb` file:

```
# spec/factories.rb

...
factory :user do
  name "Matthias Günther"
  email "matthias.guenther@wikimatze.de"
end

factory :job_offer do
  title "Padrino Engineer"
  location "Berlin"
  description "We want you ..."
  contact "recruter@awesome.de"
  time_start "0/01/2013"
  time_end "01/03/2013"
```

```
end
...
```

And now we modify our `user_spec`:

```
# spec/user_spec.rb

require &#39;spec_helper&#39;

describe "User Model" do
  let(:user) { FactoryGirl.build(:user) }
  it &#39;can be created&#39; do
    user.should_not be_nil
  end

  it &#39;fresh user should have no offers&#39; do
    user.job_offers.size.should == 0
  end

  it &#39;has job-offers&#39; do
    user.job_offers.build(FactoryGirl.attributes_for(:job_offer))
    user.job_offers.size.should == 1
  end

end
```

As you see, the job fixtures us created with FactoryGirls' `attributes_for` method. This method takes a symbol as an input and returns the attributes of the fixture as a hash.

Now, our tests are looking fine and they are still green. But we can do even better. We can remove the `FactoryGirl` expressions if we add make the following change to our `spec_helper.rb`:

```
# spec/spec_helper.rb

RSpec.configure do |conf|
  conf.include Rack::Test::Methods
  conf.include FactoryGirl::Syntax::Methods
end
```

Now we can change our test to:

```
# spec/app/models/user_spec.rb

require &#39;spec_helper&#39;

describe "User Model" do
  let(:user) { build(:user) }
  it &#39;can be created&#39; do
    user.should_not be_nil
  end

  it &#39;fresh user should have no offers&#39; do
    user.job_offers.size.should == 0
  end

  it &#39;has job-offers&#39; do
    user.job_offers.build(attributes_for(:job_offer))
    user.job_offers.size.should == 1
  end

end
```

Registration and Login

In traditional frameworks you would generate a user with a `user` model and a `users_controller` with the actions `new`, `create`, `update`, and `delete`. And you can't forget about security these days it would be nice to to have something at hand to save atemail the end we would need to find a method of safely storing the password for the user.

Of course, we could use you don't have to reinvent the wheel you can use Padrino's beautiful Admin interface for your user authentication to prevent us from reinventing the wheel. But with that we won't learn the basics and as you will see in this chapter you can make a lot of mistakes. So step into the part of creating user, sending confirmation mails, and understanding how sessions are managed in Padrino.

Extending the User Model

Before we are going to build the controller and the sign-up form for our application we need to specify the data each user has.

```
Name: String
Email: String
Password: String
```

Recording from chapter "???" we only need to add the `Password` fields to the user table:

Let's create the migration:

```
$ padrino g migration AddRegistrationFieldsToUsers

apply  orms/activerecord
create  db/migrate/004_add_registration_fields_to_users.rb
```

And write the fields into the migration file:

```
# db/migrate/004_add_registration_fields_to_users.rb

class AddRegistrationFieldsToUsers < ActiveRecord::Migration

  @fields = [:password]

  def self.up
    change_table :users do |t|
      @fields.each { |field| t.string field}
    end
  end

  def self.down
    change_table :users do |t|
      @fields.each { |field| remove_column field}
    end
  end
end
```

Ok, run the migrations:

```
$ padrino rake ar:migrate
```

Validating attributes

Before we are going to implement what we think, we are going to write **pending** specs:

```
# spec/app/models/user_spec.rb

require &#39;spec_helper&#39;

describe "User Model" do
  ...

  pending(&#39;no blank name&#39;)
  pending(&#39;no blank email&#39;)

  describe "passwords" do
    pending(&#39;no blank password&#39;)
    pending(&#39;no blank password_confirmation&#39;)
  end

  describe "when name is already used" do
    pending(&#39;should not be saved&#39;)
  end

  describe "email address" do
    pending(&#39;valid&#39;)
    pending(&#39;not valid&#39;)
  end
end
```

(The *pending* word is optional. It is enough to write pending tests only in the form `it "test this"` and leaving the do/end block away).

Before writing code to pass these specs, we need to add the `password` field to our factory:

```
# spec/factories.rb
```

```
# encoding: utf-8
FactoryGirl.define do
...
  factory :user do
    name  "Matthias Günther"
    email "matthias.guenther@wikimatze.de"
    password "octocat"
  end
end
```

Use the encoding property to allow special symbols from Germany - you have to add them in your files where they may occur. Let's implement the first pending test that a user can't have an empty name:

```
# spec/app/models/user_spec.rb
...

it &#39;have no blank name&#39; do
  user.name = ""
  user.save.should be_false
end
```

If we run the test we get the following error:

```
$ rspec spec

Failures:

  1) User Model have no blank name
     Failure/Error: user.save.should be_false
       expected: false value
            got: true
     # ./spec/app/models/user_spec.rb:20:in `block (2 levels) in <top (required)>

Finished in 0.42945 seconds
10 examples, 1 failure, 5 pending

Failed examples:

rspec ./spec/app/models/user_spec.rb:18 # User Model have no blank name
```

To make this test pass we need to validate the `email` property in our

user model with the help of the presence option:

```
# app/models/user.rb
class User < ActiveRecord::Base
  validates :name, :presence => true

  has_many :job_offers
end
```

As an exercise, Please write the validates for `email` and `password` on your own. Please consider that the `password_confirmation` attribute can be create with the `:confirmation => true` option to the `validates :password` setting.

We don't want to have duplicated names in our application. To simply test this we need as second user with the same name. In order to create a second user with we need to have another mail address. In order to write the test for it, we need to extend or factory with the sequence function:

```
# spec/factories
FactoryGirl.define do
  sequence(:email){ |n| "matthias.guenther#{n}@wikimatze.de"}

  factory :user do
    name  "Matthias Günther"
    email
    password "foo"
  end
  ...
end
```

Whenever you build a new `user` fixture the value `email_number` is incremented and gives you so a fresh user with a unique email address. You can write a test for this ability in the following way:

```
# spec/app/models/user_spec.rb
describe "when name is already used" do
  let(:user_second) { build(:user) }
```

```
    it &#39;should not be saved&#39; do
       user_second.save.should be_false
    end
  end
```

To make the test green you have to use the uniqueness validation. All what it does is to validates that the attribute's value is unique before it gets saved.

```
  # app/models/user.rb
  class User < ActiveRecord::Base
    validates :name, :email, :password, :presence => true
    validates :name, :uniqueness => true

    has_many :job_offers
  end
```

Now this test is fixed. Next we are going to implement the validation for the email field:

```
  # spec/app/models/user_spec.rb
  ...

  describe "email address" do
    it &#39;valid&#39; do
      addresses = %w[thor@marvel.de hero@movie.com]
      addresses.each do |email|
        user_second.email = email
        user_second.name= email
        user_second.should be_valid
      end
    end

    it &#39;not valid&#39; do
      addresses = %w[spamspamspam.de heman,test.com]
      addresses.each do |email|
        user_second.email= email
        user_second.should_not be_valid
      end
    end
  end
  ...
```

We can test the correctness of the `email` field with a regular expression. First we are going to define a regular expression and use the format validation which takes our regular expression against which the field will be tested.

```
# app/models/user.rb
class User < ActiveRecord::Base
...
  VALID_EMAIL_REGEX = /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i
  validates :email, format: { with: VALID_EMAIL_REGEX }
...
end
```

I> ## Regular Expressions I> I> Regular expressions are your first tool when you I> need to match certain parts (or whole) strings against a predefined pattern. The drawback of using them is that I> you have to learn a formal language to define your patterns. I can highly I> recommend you the Rubular tool for training and trying out the expression you I> want to use. It make it very easy to build and test your patterns against test data.

Users Controller

Since we already have a model for potential users of our platform, it's time to create a controller for them. We are creating in a first step our users controller four our sign up form with only one action:

```
$ padrino g controller Users get:new
  create  app/controllers/users.rb
  create  app/helpers/users_helper.rb
  create  app/views/users
   apply  tests/rspec
  create  spec/app/controllers/users_controller_spec.rb
```

The new thing about the controller command above is the `get:new` option. This will create an URL rout `:new` to `users/new`.

Sign Up Form

The stage is set: We have the model with the tested constraints, and a controller for the user which handles the action. Time to create a sign up form for getting new users on our platform. For this case we can use the `form_for` helper. This method takes an object as its input and creates a form using the attributes of the object. We need this to save/edit the attributes of the model in our controller. Create a new erb file under the users view:

```
# app/views/users/new.erb
<h1>Registration</h1>

<% form_for(@user, &#39;/users/create&#39;) do |f| %>
  <%= f.label :name %>
  <%= f.text_field :name %>
  <%= f.label :email %>
  <%= f.text_field :email %>
  <%= f.label :password %>
  <%= f.password_field :password %>
  <%= f.label :password_confirmation %>
  <%= f.password_field :password_confirmation %>
  <p>
  <%= f.submit "Create", :class => "btn btn-primary" %>
  </p>
<% end %>
```

There is a lot of stuff going on -- let's break it down:

- `form_for`: Is part of Padrino's Form Builder and allows you to create standard input fields based on a model. The first argument to the function is an object (mostly a model), the second argument is an string (the action to which the form should be sent after a submit), and the third parameter are settings in form of an hash which aren't used in this example. The part `action="/users/create"` says, that we want to use the `create` action to the `users` controller with the `create` action.

- `f.label` and `f.text`: Will a label and text field for the attributes of your model.
- `f.password_field`: Constructs a password input, where the input is marked with stars, from the given attribute of the form.
- `f.submit`: Take an string as an caption for the submit button and options as hashes for additional parameter (for example `:class => 'long'`).

This form above will be rendered in the following HTML:

```
<form method="post" action="/users/create" accept-charset="UTF-8">  <label for="u
  <input id="user_name" name="user[name]" type="text" />

  <label for="user_email">Email: </label>
  <input id="user_email" name="user[email]" type="text" />

  <label for="user_password">Password: </label>
  <input id="user_password" name="user[password]" type="password" />

  <label for="user_password_confirmation">Password confirmation: </label>
  <input id="user_password_confirmation" name="user[password_confirmation]" type=

  <p>
  <input class="btn btn-primary" value="Create" type="submit" />
  </p>
</form>
```

User Controller Signup Actions

We need to make sure to have the right mappings for the `/login` route in the actions in our controller:

```
# app/controllers/users.rb

JobVacancy::App.controllers :users do

  get :new, :map => "/login" do
    @user = User.new
    render &#39;users/new&#39;
```

```
    end

  end
```

So far so good, feel free to visit http://localhost:3000/login. Until now we are not saving the inputs of the user. And what about the mistakes a user makes during his input? How can we display any mistakes a user is making and preserve the things he already typed in?

If you remember of section [TBD HAVE TO LOOK UP WHERE USER CREATE/VALIDATE WAS CALLED] we can use this method for validation before we are going to save it. Before doing two steps at a time let's code the `create` action which saves the new registered user without going into error validation.

```
# app/controllers/users.rb

post :create do
  @user = User.new(params[:user])
  @user.save
  redirect(&amp;#39;/&amp;#39;)
end
```

Let's go through the new parts:

- `User.new(params[:users])` : Will create a new user object based on the information of the form attributes of the `@user` model which which were part of the form of the `views/users/new.erb` page.
- `@user.save`: Will save the user in the database.
- `redirect`: Will redirect the user to the root directory of our app.

If you send the form without any inputs, you will see that you are redirected into the root of your app. You can't figure out what's wrong, but luckily we have logs:

```
DEBUG -  (0.1ms)  begin transaction
DEBUG - User Exists (0.3ms)  SELECT 1 AS one FROM "users" WHERE "users"."name" =
DEBUG - User Exists (0.2ms)  SELECT 1 AS one FROM "users" WHERE "users"."email" =
```

```
DEBUG -  (0.2ms)  rollback transaction
DEBUG -     POST (0.0162ms) /users/create - 303 See Other
DEBUG - TEMPLATE (0.0004ms) /page/home
DEBUG - TEMPLATE (0.0002ms) /application
DEBUG -      GET (0.0057ms) / - 200 OK
DEBUG -      GET (0.0005ms) application.css?1365616902 - 200 OK
DEBUG -      GET (0.0003ms) application.js?1365616902 - 200 OK
DEBUG -      GET (0.0017ms) /favicon.ico - 404 Not Found
```

The part with the `rollback transaction` means, that user was not saved. Why? Because he violated the validation of our user model. Try to create an `User.new` model in the console and call the `.errors` method on. You should see something like:

```
=> #<ActiveModel::Errors:0x9dea518 @base=#<User id: nil, name: nil, email: nil, c
    updated_at: nil, password: nil>, messages{:name=>["can&#39;t be blank"],
    :password=>["is too short (minimum is 5 characters)", "can&#39;t be blank"],
    :email=>["can&#39;t be blank", "is invalid"]}
```

We can use this information to display the errors in our form for the user to let him know what they did wrong. If you want a dirty and quick solution, you can use the `form.error_messages`, which simply can be put at the front of our form:

```
# views/users/new.erb


<% form_for(@user, &#39;/users/create&#39;) do |f| %>
  ...
  <%= f.error_messages %>
  ...
<% end %>
```

It counts the number of errors (`@user.errors.count`) and is looping through all field with their error messages. But this will result in a big box with a bunch of error messages like the following one:

```
5 errors prohibited this User from being saved
There were problems with the following fields:
```

```
Name can&#39;t be blank
Password is too short (minimum is 5 characters)
Password can&#39;t be blank
Email can&#39;t be blank
Email is invalid
```

This isn't something we want to ship to our customers.

Let's change this by using error_message_on method which returns a string containing the error message attached to the method on the object:

```
# views/users/new.erb

<% form_for(@user, &#39;/users/create&#39;) do |f| %> <%= f.label :name %>
  <%= f.text_field :name %>
  <%= error_message_on @user, :name %>
  <%= f.label :email %>
  <%= f.text_field :email %>
  <%= error_message_on @user, :email %>
  <%= f.label :password %>
  <%= f.password_field :password %>
  <%= error_message_on @user, :password %>
  <%= f.label :password_confirmation %>
  <%= f.password_field :password_confirmation %>
  <%= error_message_on @user, :password_confirmation %>
  <p>
  <%= f.submit "Create", :class => "btn btn-primary" %>
  </p>
<% end %>
```

We can do better and make the error text red. Let's add the `:class` at the of the `error_message_on` method with the help of the text-error class from bootstrap and using the `:prepend` option which add text to before

displaying the field error:

```
    # views/users/new.erb


    <% form_for(@user, &#39;/users/create&#39;) do |f| %>
      <%= f.label :name %>
      <%= f.text_field :name %>
      <%= error_message_on @user, :name, :class => "text-error", :prepend => "The name
      <%= f.label :email %>
      <%= f.text_field :email %>
      <%= error_message_on @user, :email, :class => "text-error", :prepend => "The ema
      <%= f.label :password %>
      <%= f.password_field :password %>
      <%= error_message_on @user, :password, :class => "text-error", :prepend => "The
      <%= f.label :password_confirmation %>
      <%= f.password_field :password_confirmation %>
      <%= error_message_on @user, :password_confirmation, :class => "text-error" %>
      <p>
      <%= f.submit "Create", :class => "btn btn-primary" %>
      </p>
    <% end %>
```

If you fill out the form with complete valid parameters and watch your log again, you can see the following log:

```
    DEBUG -  (0.2ms)  begin transaction
    DEBUG - User Exists (0.3ms)  SELECT 1 AS one FROM "users" WHERE "users"."name" =
    DEBUG - User Exists (0.2ms)  SELECT 1 AS one FROM "users" WHERE "users"."email" =
    DEBUG - SQL (0.2ms)  INSERT INTO "users" ("created_at", "email", "name", "passwor
    (?, ?, ?, ?, ?)  [["created_at", 2013-04-10 20:09:10 +0200], ["email", "admin@job
    ["name", "Testuser"], ["password", "example"], ["updated_at", 2013-04-10 20:09:10
    DEBUG -  (174.1ms)  commit transaction
    DEBUG -     POST (0.1854ms) /users/create - 303 See Other
    DEBUG - TEMPLATE (0.0004ms) /page/home
    DEBUG - TEMPLATE (0.0002ms) /application
    DEBUG -      GET (0.0058ms) / - 200 OK
    DEBUG -      GET (0.0006ms) application.css?1365617350 - 200 OK
```

```
DEBUG -      GET (0.0002ms) application.js?1365617350 - 200 OK
DEBUG -      GET (0.0018ms) /favicon.ico - 404 Not Found
```

Remember that have an eye on your logs can help you to see what's going on in your back-end when you can't see it in the front-end of your app.

I> ## What are VALUES (?, ?, ?, ?, ?) in a SQL insert query? I> I> These form of inserting data in your database is known as parameterized queries. A parameterized query is a query I> in which placeholders are used for parameters and the I> parameter values are supplied at execution time. The most important reason to use parameterized queries is to avoid I> SQL injection attacks. SQL injection means that SQL statements are I> injected into input fields in order to drop tables or getting access on user related data.

Emails

Padrino uses the Padrino Mail gem for sending mail. For simplification, we are using SMTP with Gmail. First of all we need to give our application the settings for setting mails in the main configuration file of our application `app.rb`:

```
# app/app.rb

module JobVacancy
  class App < Padrino::Application
    ...
    set :delivery_method, :smtp => {
      :address => &#39;smtp.gmail.com&#39;,
      :port => 587,
      :user_name => &#39;<your-gmail-account-address>&#39;,
      :password => &#39;<secret>&#39;,
      :authentication => :plain,
```

```
      }
    end
  end
```

Let's get through all the different options:

- `:delivery_method`: Defines the delivery method. Possible values are :smtp (default), :sendmail, `:test` (no mails will be send), and :file (will write the contents of the email in a file).
- `:address`: The SMTP mail address.
- `:port`: The port of the mail address.
- `:user_name`: The name of the SMTP address.
- `:password`: The password of your SMTP address.
- `:authentication`: Specify if your mail server requires authentication. The default setting is plain meaning that the password is not encrypted. `:login` will send the password Base64 encoded, :cram_md5 is a challenge/response authentication mechanism .
- `:domain`: This key is set up for HELO checking.

Prior Padrino 0.10.7 the `:enable_starttls_auto => true` was changeable. This is option is now always on true in Padrino >= 0.11.1.

This is now the default delivery address unless it is overwritten in an individual mail definition. We won't test the email functionality to this point because the *Mailer gem* is already tested.

Quick Mail Usage

To send a first simple "Hallo" message we create an email block directly in our user controller:

```
# app/controllers/users.rb

post :create do
  @user = User.new(params[:user])
  if @user.save
    email do
```

```
        from "admin@job-vacancy.de"
        to "lordmatze@gmail.com"
        subject "Welcome!"
        body "hallo"
      end
      redirect(&amp;#39;/&amp;#39;)
    else
      render &amp;#39;users/new&amp;#39;
    end
  end
```

Now start the app, go to the URL `http://localhost:3000/login` , and register a fresh user. You can check the log if the mail was send or you just "feel" a slow down in your application because it takes a while before the mail is send::

```
DEBUG - Sending email to: lordmatze@gmail.com
Date: Sun, 14 Apr 2013 09:17:38 +0200
From: admin@job-vacancy.de
To: lordmatze@gmail.com
Message-ID: <516a581243fb3_498a446f81c295e3@mg.mail>
Subject: Welcome!
Mime-Version: 1.0
Content-Type: text/plain;
 charset=UTF-8
Content-Transfer-Encoding: 7bit

Hallo
```

Mailer

We could go on and parametrized our email example above, but this would mean that we have email code directly into our controller code. We can do better by wrapping up the logic into an object and let it handle the action.

Padrino mailer has the `mailer` command to create customized mailer for every purpose we want to use. Let's create the registration mailer:

```
$ padrino g mailer Registration registration_email
  create  app/mailers/registration.rb
```

```
    create  app/views/mailers/registration
```

Let's break it down:

- `mailer`: The command to create a custom mailer. Inside a mailer you can define the name of your mailer object and it's different templates. The name of our first email is `registration_email`.
- `Registration`: Name of the mailer.

Now we let's look into the `registration.rb` file:

```
# app/mailers/registration.rb

JobVacancy:App.mailer :registration do
  email :registration_email do
    # Your mailer goes here
  end
end
```

The generated comment `# Your mailer goes here` says what you have to do. So let's remove the code from our `users` controller and move it to this place.

```
# app/mailers/registration.rb

JobVacancy::App.mailer :registration do
  email :registration_email do
    from "admin@job-vacancy.de"
    to "lordmatze@gmail.com"
    subject "Welcome!"
    body "Hallo"
  end
end
```

Now we can use the *deliver* method to call our `:registration` mailer with it's template `:registration_email`:

```
# app/controller/users.rb

...
post :create do
```

```
    @user = User.new(params[:user])
    if @user.save
      deliver(:registration, :registration_email)
      redirect(&amp;#39;/&amp;#39;)
    else
      render &amp;#39;users/new&amp;#39;
    end
  end
  ...
```

I> Difference between Padrino's Mailer methods email and deliver I> I> The email method is I> has the parameters `mail_attributes = {}, &block` . That means the you write emails directly I> `JobVacancy.email(:to => '...', :fro` or use the block syntax I> `JobVacancy.email do ... end` . In comparison to this is the I> deliver method. I> It has `mailer_name, message_name, *attributes` as attributes. In order to use this you always to create a Mailer I> for them. If you want to use very simple mails in your application, prefer to use the email method. But if you I> have templates with a much more complex layout in different formats (plain, HTML), the deliver method is the I> best fit.

Instead of writing only a simple "Hallo" in our email we would like to give more input. First we need to write an template and then use the `render` method in our registration mailer. Let's define the registration template:

```
# app/views/mailers/registration/registration_email.plain.erb

Hi ...,

we are glad to have you on our platform. Feel free to post jobs and find the righ

Your Job Vacancy!
```

And now we make sure that we are rendering this template in our registration mailer:

```
# app/mailers/registration.rb
```

```
JobVacancy::App.mailer :registration do
  email :registration_email do
    from "admin@job-vacancy.de"
    to "lordmatze@gmail.com"
    subject "Welcome!"
    render &#39;registration/registration_email&#39;
    content_type :plain
  end
end
```

If you are sure that you only want to send plain text mail, you can leave the `plain` extension away but making it explicit will make it clear what you want to do.

To make our email more personal we want to add the name of our freshly registered user to our email template. In order to do this we need to use enable the `locals` option.

```
# app/mailers/registration.rb

JobVacancy::App.mailer :registration do
  email :registration_email do |name, email|
    from "admin@job-vacancy.de"
    to email
    subject "Welcome!"
    locals :name => name
    render &#39;registration/registration_email&#39;
    content_type :plain
  end
end
```

This options enables a hash which we be used in the email template. Now we need to pass the name to the call of our method in our `users` controller:

```
# app/controllers/users.rb
...

post :create do
  @user = User.new(params[:user])

  if @user.save
```

```
      deliver(:registration, :registration_email, @user.name)
      redirect(&amp;#39;/&amp;#39;)
    else
      render &amp;#39;users/new&amp;#39;
    end
  end
```

And update our template with the name variable:

```
# app/views/mailers/registration/registration_email.plain.erb

Hi <%= name %>,

we are glad to have you on our platform. Feel free to post jobs and find the right

Your Job Vacancy!
```

Next we want to add a PDF which explains the main business needs to our page. For this purpose we create add the `welcome.pdf` into the `/app/assets/` folder. We can attach files (images, PDF, video) with the add_file method which takes a filename and the content as hash elements as arguments.

```
# app/mailers/registration.rb
...

email :registration_email do |name, email|
  from "admin@job-vacancy.de"
  to email
  subject "Welcome!"
  locals :name => name, :email=> email
  render &amp;#39;registration/registration_email&amp;#39;
  add_file :filename => &amp;#39;welcome.pdf&amp;#39;, :content => File.open("#{Padrino.r
end
```

Please correct me if there is a better way to get to the asset folder but that is all of what I've found.

During writing this chapter I experiment with the `content_type` option.

If you set the `content_type` to plain you will get the attachment based as binary code directly into your mail. Please put the `content_type :plain` into the `registration` mailer. If the mail will be send you can see something like this in your logs:

```
  DEBUG - Sending email to: lordmatze@gmail.com
Date: Thu, 18 Apr 2013 18:34:15 +0200
From: admin@job-vacancy.de
To: lordmatze@gmail.com
Message-ID: <5170208754967_70f748e80108323a@mg.mail>
Subject: Welcome!
Mime-Version: 1.0
Content-Type: text/plain
Content-Transfer-Encoding: 7bit

----==_mimepart_517020874676e_70f748e8010829d8
Date: Thu, 18 Apr 2013 18:34:15 +0200
Mime-Version: 1.0
Content-Type: text/plain;
 charset=UTF-8
Content-Transfer-Encoding: 7bit
Content-ID: <5170208753fd5_70f748e8010831c1@mg.mail>

Hi Bob,

we are glad to have you on our platform. Feel free to post jobs and find the ri

Your Job Vacancy!

----==_mimepart_517020874676e_70f748e8010829d8
Date: Thu, 18 Apr 2013 18:34:15 +0200
Mime-Version: 1.0
Content-Type: application/pdf;
 charset=UTF-8;
 filename=welcome2.pdf
Content-Transfer-Encoding: base64
Content-Disposition: attachment;
 filename=welcome2.pdf
Content-ID: <517020874ba76_70f748e80108301@mg.mail>

JVBERi0xLjQKJcOkw7zDtsOfCjIgMCBvYmoKPDwvTGVuZ3RoIDMgMCBSL0Zp
bHRlci9GbGF0ZURlY29kZT4+CnN0cmVhbQp4nDPQM1Qo5ypUMABCM0MjBXNL
...
# You don&#39;t want to read four pages of strange characters ...
```

```
...

----==_mimepart_517020874676e_70f748e8010829d8--
```

I> ## MIME? I> I> MIME stands for "Multipurpose Internet Mail Extensions" and they specify additional attributes to email headers like I> the content type and define transfer encodings which can be used to present a higher encoded file (e.g. 8-bit) I> with the 7-bit ASCII character set. This makes it possible to put non-English characters in the message header like I> the subject. The goal of the MIME definition was that existing email servers had nothing to change in order to use I> MIME types. This means that MIME headers are optional for plain text emails and so even none MIME messages can be I> read correctly by a clients being able to read MIME encoded messages.

Sending Email with Confirmation Link

The basic steps for implementing the logic of email confirmation are the following:

- we need to add the *confirmation_code* and *confirmation* attributes in our user model.
- create a controller method for our user model that expects a user id and confirmation code, looks up the user, checks the code in the parameter matches the code saved in our database and clears the code after confirmation.
- create an action that maps to our new controller method (e.g. `/confirm/<use`).
- create an mailer template which takes the user as a parameter and use the *confirmation code* of the user to send a mail containing a link to the new route in our controller.
- create an **observer** for our user model. if the email of the user needs to be modified or a record is created we need to create a

confirmation code, set it in the model and clear the confirmation flag. after that we need to trigger our mailer.
- create a helper method which allows views to check if the current user is confirmed.
- protect our controller methods and views to prevent security issues.

I> ## Why Confirmation Mail I> I> Check that the user actually signed up for the account and actually wants it. This also helps you from spamming your I> platform is going to be floated with billions of users. Another usage of this information is to give your users a I> chance to change their password and/or stay in contact with them to inform them about updates.

Add Confirmation Code and Confirmation Attributes to the User Model

Create a good migration which fits to the task we want to do:

```
padrino g migration add_confirmation_code_and_confirmation_to_users
   apply  orms/activerecord
  create  db/migrate/005_add_confirmation_code_and_confirmation_to_users.rb
```

Now let's add the fields to a migration:

```
# db/migrate/005_add_confirmation_code_and_confirmation_to_users.rb

class addconfirmationcodeandconfirmationtousers < activerecord::migration
  def self.up
    change_table :users do
      t.string :confirmation_code
      t.boolean :confirmation, :default => false
    end
  end

  def self.down
```

```
      change_table :users do
        remove_column :confirmation_code, :confirmation
      end
    end
  end
```

We added the `:default` option which sets the confirmation for every user to false if a new one is registered. now let's migrate our production and test database to this new event:

```
$ padrino ar:migrate
$ padrino ar:migrate -e test
```

My Tests are Slow ...

During writing this book I discovered various strange behavior for my tests because I was writing data into my test database. So the tests weren't really reliable because some worked only when the database is fresh with no preexisting entries. One solution would be to clean up the database before each run:

```
$ sqlite3 db/job_vacancy_test.db
  SQLite version 3.7.13 2012-06-11 02:05:22
  Enter ".help" for instructions
  Enter SQL statements terminated with a ";"
  sqlite> DELETE FROM users;
  sqlite> .quit
```

But after this my tests were running very slow:

```
$ rspec spec
...

Finished in 1.61 seconds
25 examples, 0 failures
```

Running them again make them a little bit faster:

```
$ rspec spec
...
```

```
Finished in 0.77209 seconds
25 examples, 0 failures
```

Why? Because we are hitting the database and our tests slow. Please consider code like the following:

```
# spec/app/models/user_spec.rb

describe "when name is already used" do
  it &#39;should not be saved&#39; do
    user.save
    user_second.name = user.name
    user_second.should_not be_valid
  end
end

describe "when email address is already used" do
  it &#39;should not save an user with an existing address&#39; do
    user.save
    user_second.email = user.email
    user_second.save.should be_false
  end
end
```

We can use mocks the saving operation away. The benefit of mocks are that you create the environment you want to test and don't care about all the preconditions to make this test possible.

Consider the following code example:

```
post :create do
  user = User.find_by_email(params[:email])

  if user && user.confirmation && user.password == params[:password]
    redirect &#39;/&#39;
  else
    render &#39;/sessions/new&#39;
  end
end
```

In order to test the condition `if user && user.confirmation && user.password == para`

to return the redirect we need find a User by email out of our database. A normal test would be in need to create a user, save it and giving the object the right attributes to pass it. We can use mocks to simulate this environment by creating a user out of our users factory, setting the attributes of this object and cheating our `find-by-email` method to return our factory user with the right params without actually saving our object to the database:

```
  it "should redirect if user is correct" do
    user.confirmation = true
    User.should_receive(:find_by_email).and_return(user)
    post "sessions/create", user.attributes

    last_response.should be_redirect
  end
```

The magic behind mocking is to use the should_receive and and_return flow. `Should_receive` says which method should be called and `and_return` what should be returned when the specified method is called. The line size of our tests will remain the same - you only have to write more characters :) but this will speed up your tests in the long term. With the help of mocks you keep your tests fast and robust.

Even if this will be the first application you write, when you've learned something new and this will make your life easier, go back and take your time to enhance the style and design of your application.

Controller Method and Action For Password Confirmation

When we are going to register a new user, we need to create a confirmation code like in the example above. Since this is business logic, we will put this method inside our users model. First we will write a failing test:

```
# spec/app/models/user_spec.rb
...

  describe "confirmation code" do
    let(:user_confirmation) { build(:user) }

    it &#39;should not be blank&#39; do
      user_confirmation.confirmation_code = ""
      user_confirmation.valid?.should be_false
    end
  end
```

To make this test pass we add the validates presence of ability in our user model:

```
# app/models/user.rb

class User < ActiveRecord::Base
  ...
  validates :confirmation_code, :presence => true
  ...
end
```

Next we need think of how we can set the `confirmation_code` information to our freshly created user. Instead of creating a confirmation code on our own, we want to encrypt the password by some mechanism. Luckily, we can use bcrypt gem to create our confirmation code. It is a Ruby binding for the OpenBSD bcrypt password hashing algorithm. In order to use this in our app we need to add it to our `Gemfile`:

```
# Gemfile
...

# Security
gem &#39;bcrypt-ruby&#39;, &#39;3.0.1&#39;, :require => &#39;bcrypt&#39;
```

Now let's open the console and play around with this Gem:

```
$ padrino c
=> Loading development console (Padrino v.0.11.1)
=> Loading Application JobVacancy
```

```
>> password = "Test11111134543"
=> "Test11111134543"
>> salt = "$2a$05$CCCCCCCCCCCCCCCCCCCCCC.E5YPO9kmyuRGyh0XouQYb4YMJKvyOeW"
=> "$2a$05$CCCCCCCCCCCCCCCCCCCCCC.E5YPO9kmyuRGyh0XouQYb4YMJKvyOeW"
>> BCrypt::Engine.hash_secret(password, salt)
=> "$2a$05$CCCCCCCCCCCCCCCCCCCCCC.9APD.dklRtXYdki/E3XrHiCWd/rfAFu"
```

I> ## What is a Salt? I> Salts are used in cryptography as random data to be put as addition to normal password to create a encrypted with the I> help of a one-way function. A one-way function output by some input string very easily but the other way round is I> very difficult for the computer to compute the original string from the output. I> Salts make it more difficult for hackers to get the password via rainbow tables attacks. Rainbow tables are a huge I> list of precomputed hashes for widely used password. If a hacker gets access to a password hash he then just I> compare this hash with the entries. If he finds after which he was searching he got the password for the user.

We could add these methods in the users controller but that isn't something a controller should do. We better use a callback. **Callbacks** are methods to run on a certain stage or life cycle of an object. Perfect, that's what we want let's create code for it:

```
# app/models/user.rb

class User < ActiveRecord::Base
  ... # The other validations

  before_save :encrypt_confirmation_code, :if => :registered? # our callback with

  private
  def encrypt_confirmation_code
    self.confirmation_code = set_confirmation_code
  end

  def set_confirmation_code
    require &#39;bcrypt&#39;
    salt = BCrypt::Engine.generate_salt
```

```
      confirmation_code = BCrypt::Engine.hash_secret(self.password, salt)
      normalize_confirmation_code(confirmation_code)
    end

    def normalize_confirmation_code(confirmation_code)
      confirmation_code.gsub("/", "")
    end

    def registered?
      self.new_record?
    end
  end
```

We won't test the methods under the private keyword, there is no customized business logic inside these methods. We will even not test the difficult looking `set_confirmation_code` method because there is no customized business logic inside, and BCrypt is well tested.

I> ## Why making callbacks private? I> I> It is good practice to make your callbacks private so that they can called only from inside the model and no other I> object can use these methods. Our `confirmation_code` method is public available but that is no problem, because it I> just generates an random string.

After creating the confirmation code mechanism for our user, we need to implement a authenticate which takes the confirmation code as an input and mark our user as *confirmed*. As always, let's begin with failing tests first:

```
# spec/app/models/user_spec.rb
...

describe "confirmation code" do
  let(:user_confirmation) { build(:user) }

  it &#39;should not be blank&#39; do
    user_confirmation.confirmation_code = ""
    user_confirmation.valid?.should be_false
  end
```

```
    it &amp;#39;should authenticate user with correct confirmation code&amp;#39; do
      user_confirmation.save
      confirmation_of_saved_user = User.find_by_id(user_confirmation.id)
      user_confirmation.confirmation_code = confirmation_of_saved_user.confirmation
      user_confirmation.authenticate(user_confirmation.confirmation_code).should be
    end

    it &amp;#39;confirmation should be set true after a user is authenticated&amp;#39; do
      user_confirmation.save
      confirmation_of_saved_user = User.find_by_id(user_confirmation.id)
      user_confirmation.confirmation_code = confirmation_of_saved_user.confirmation
      user_confirmation.authenticate(user_confirmation.confirmation_code).should be
      user_confirmation.confirmation.should be_true
    end

    it &amp;#39;should not authenticate user with incorrect confirmation code&amp;#39; do
      user_confirmation.authenticate("wrong").should be_false
    end
  end
```

I> ## Take care of your names!? I> I> During writing this chapter I lost a whole hour because I had method with the same name as the `confirmati` I> field. When I wanted to check `@user.confirmation_code` it always called the `confirmation_code` method which return I> a new confirmation code. I was thinking for a long time that it returned the attribute and was wondering what's going I> on. A couple of pry sessions showed me nothing since I'm expected to be right. After I went I> to the toilet I started another pry session and out of sudden I discovered my naming problem. I> I> Lesson learned: Breaks are great!

Before going on we need to update our `factory` for the test with the confirmation code field::

```
  # spec/factories.rb

  # encoding: utf-8
  FactoryGirl.define do
    ...
    sequence(:confirmation_code){ "1" }
    sequence(:id){ |n| n }
```

```
  factory :user do
    id
    name
    email
    password "octocat"
    confirmation_code
  end
  ...
end
```

We are making the `confirmation_code` with the value of 1 static because this make it easier for us to test the code. Here is now the code that makes our tests green:

```
# app/models/user.rb

class User < ActiveRecord::Base
  ...

  def authenticate(confirmation_code)
    return false unless @user = User.find_by_id(self.id)

    if @user.confirmation_code == self.confirmation_code
      self.confirmation = true
      self.save
      true
    else
      false
    end
  end
  ...
end
```

Since our tests of our user model are now green, let's write tests for our /confirm route:

```
# spec/app/controllers/users_controller.rb

describe "GET confirm" do
let(:user) { build(:user) }
  it "render the &#39;users/confirm&#39; page if user has confirmation code" do
    user.save
    get "/confirm/#{user.id}/#{user.confirmation_code.to_s}"
```

```
      last_response.should be_ok
    end

    it "redirect the :confirm if user id is wrong" do
      get "/confirm/test/#{user.confirmation_code.to_s}"
      last_response.should be_redirect
    end

    it "redirect to :confirm if confirmation id is wrong" do
      get "/confirm/#{user.id}/test"
      last_response.should be_redirect
    end
  end
```

To make this pass, we implement the following code:

```
  # app/controllers/users.rb
  ...

get :confirm, :map => "/confirm/:id/:code" do
  redirect(&#39;/&#39;) unless @user = User.find_by_id(params[:id])
  redirect(&#39;/&#39;) unless @user.authenticate(params[:code])
  render &#39;users/confirm&#39;
end
```

Mailer Template for Confirmation Email

If we are lazy we could add our confirmation email into the registration mailer. But if you think clearly, these are two things that have nothing to do with each other. So let's train our memory and create another mailer:

```
  $ padrino g mailer Confirmation confirmation_email
    create  app/mailers/confirmation.rb
    create  app/views/mailers/confirmation
```

Now let's fill out the confirmation mailer:

```
  # app/mailers/confirmation.rb

  JobVacancy::App.mailer :confirmation do
    CONFIRMATION_URL = "http://localhost:3000/confirm"
```

```
    email :confirmation_email do |name, email, id, link|
      from "admin@job-vacancy.de"
      subject "Please confirm your account"
      to email
      locals :name => name, :confirmation_link => "#{CONFIRMATION_URL}/#{id}/#{link
      render &#39;confirmation/confirmation_email&#39;
    end
  end
```

Fill the email template with "confirmation-link-life":

```
# app/views/mailers/confirmation/confirmation_email.plain.erb


Hi <%= name %>,


to take fully advantage of our platform you have to follow the following link:


<%= confirmation_link %>


Enjoy the possibility to find the right people for your jobs.
```

And call this method to our users controller:

```
  # app/controllers/users.rb

  post :create do
    @user = User.new(params[:user])

    if @user.save
      deliver(:registration, :registration_email, @user.name, @user.email)
      deliver(:confirmation, :confirmation_email, @user.name,
              @user.email,
              @user.id,
              @user.confirmation_code)
      redirect(&#39;/&#39;)
    else
      render &#39;users/new&#39;
    end
  end
```

Observer

In the chapter[TBD done find out how to reference to other chapters] we used a callback to build the functionality with the confirmation code generation and sending. If you think clearly we have flaws in our design:

1. The controller is sending mails but this is not the responsibility of it.
2. Our user model is blown up with authentication code.

I> ## Observers vs. Callbacks I> I> Observers are a design pattern where an object has a list of its I> dependents called observers, and notifies them automatically if its state has changed by calling one of their methods. I> Observers means to be decoupling responsibility. They can I> serve as a connection point between your models and some other functionality of another subsystem. Observers "lives" I> longer in your application and can be attached/detached at any time. I> Callbacks life shorter - you pass it to a function to be called only once. I> Rule of the thumb: When you use callbacks with code that isn't directly related to your model, you better put this I> into an observer.

Here is a rough plan what we want to do:

- Create an observer in the models folder.
- Register the observer in app/app.rb.
- Attach the observer to the model so that the user model automatically calls the method of the observer.

Let's create the observer with the name `user_observer` in the models folder

```
# app/models/user_observer.rb
```

```
class UserObserver < ActiveRecord::Observer
... # put in here the private methods of the users model
end
```

(Sadly, Padrino hasn't a generate command for this but I'm having this on my list to create a pull request for this feature.)

So we are defining our user observer with extends from the ActiveRecord::Observer. Inside this class we can define any callbacks for each action we want to use. The most commons ones are `before_<action>` and `after_<action>` where `<action>` is the ActiveRecord trigger method like save, update, delete, show, or get. To see what we can move out of the user model let's have a look inside this model:

```
# app/models/user.rb

class User < ActiveRecord::Base
  ... # The other validations

  before_save :encrypt_confirmation_code, :if => :registered?

  private
  def encrypt_confirmation_code
    self.confirmation_code = set_confirmation_code
  end

  def set_confirmation_code
    require &#39;bcrypt&#39;
    salt = BCrypt::Engine.generate_salt
    confirmation_code = BCrypt::Engine.hash_secret(self.password, salt)
    normalize_confirmation_code(confirmation_code)
  end

  def registered?
    self.new_record?
  end

  def normalize_confirmation_code(confirmation_code)
    confirmation_code.gsub("/", "")
  end
end
```

And refactor the code above into our observer:

```
# app/models/user_observer.rb

class UserObserver < ActiveRecord::Observer

  private
  def encrypt_confirmation_code(user)
    user.confirmation_code = set_confirmation_code(user)
  end

  def set_confirmation_code(user)
    require &#39;bcrypt&#39;
    salt = BCrypt::Engine.generate_salt
    confirmation_code = BCrypt::Engine.hash_secret(user.password, salt)
    normalize_confirmation_code(confirmation_code)
  end

  def normalize_confirmation_code(confirmation_code)
    confirmation_code.gsub("/", "")
  end
end
```

So far so good, but we also need to remove the callback `before_save :encrypt_cor` and we need also to transfer this logic:

```
# app/models/user_observer.rb

class UserObserver < ActiveRecord::Observer
...
  def before_save(user)
    if user.new_record?
      encrypt_confirmation_code(user)
      JobVacancy.deliver(:registration, :registration_email, user.name, user.emai
    end
  end
...
end
```

If we have a fresh registered user we create an confirmation code and send him an welcome mail. Hmm, but what about the confirmation of our user? Right, we need to add an `after_save` method which send the

confirmation code to the user:

```
# app/models/user_observer.rb
...

def after_save(user)
  JobVacancy.deliver(:confirmation, :confirmation_email, user.name,
                     user.email,
                     user.id,
                     user.confirmation_code) unless user.confirmation
end
```

We have cleaned up our design to this time. Before that if a user update his profile a new confirmation code will be send. We fixed this is with the `unless user.confirmation` line which means as long as the user is not confirmed, send him the confirmation code. We haven't any test for this kind and if you are curious how to do it, feel free to write a test for this and modify the observer code. I haven't found a way to test this - maybe you use the mock_model for your tests! We cleaned up our users controller from sending mail and this is the best solution, because in it's heart a controller just talks to the model and passing the ball to the right direction after an event.

The last step we need to do is to register our observer in the `app.rb` and disable the observer for our specs

```
# app/app.rb.
module JobVacancy
  class App < Padrino::Application
    ...
    # Activating the user_observer
    ActiveRecord::Base.add_observer UserObserver.instance
    ...
  end
end

# spec/spec_helper.rb

RSpec.configure do |conf|
  ...
```

```
    ActiveRecord::Base.observers.disable :all # <-- Turn &#39;em all off!
    ...
  end
```

If you want to have an observer test[2], you can use the following one:

```
  # spec/app/models/user_observer_spec.rb
  require &#39;spec_helper&#39;

  describe "UserObserver" do
    let(:user) { build(:user)}
    before do
      @observer = UserObserver.instance
      @model = User
    end

    it &#39;creates Mail::Message object before save&#39; do
      @observer.before_save(user).should be_instance_of(Mail::Message)
    end

    it &#39;do not create Mail::Message if user already exist&#39; do
      @observer.before_save(@model.first).should be_nil
    end

    it &#39;creates Mail::Message object after save&#39; do
      @observer.after_save(user).should be_instance_of(Mail::Message)
    end

  end
```

But during writing this book I became different testing results when using `bundle exec rake spec` and `bundle exec rspec spec` and to go on with the book, I removed the test and disabled all observers for the application.

Sessions

Now that our users have the possibility to register and confirm on our page, we need to make it possible for our users to sign in. For

handling login, we need to create a session controller:

```
$ padrino g controller Sessions new create destroy
  create  app/controllers/sessions.rb
  create  app/helpers/sessions_helper.rb
  create  app/views/sessions
   apply  tests/rspec
  create  spec/app/controllers/sessions_controller_spec.rb
```

We made a mistake during the generation - we forget to add the right action for our request. Before making the mistake to delete the generated files by hand with a couple of `rm's`, you can run a generator to destroy a controller:

```
$ padrino g controller Sessions -d
  remove  app/controllers/sessions.rb
  remove  app/helpers/sessions_helper.rb
  remove  app/views/sessions
   apply  tests/rspec
  remove  spec/app/controllers/sessions_controller_spec.rb
```

And run the generate command with the correct actions:

```
$ padrino g controller Sessions get:new post:create delete:destroy
  create  app/controllers/sessions.rb
  create  app/helpers/sessions_helper.rb
  create  app/views/sessions
   apply  tests/rspec
  create  spec/app/controllers/sessions_controller_spec.rb
```

Our session controller is naked:

```
# app/controllers/sessions_controller.rb

JobVacancy:App.controllers :sessions do

  get :new, :map => "/login" do
  end

  post :create do
  end
```

```
    delete :destroy do
    end
  end
```

So far so good before going on to write our tests first before we start with the implementation:

```
  # spec/app/controllers/sessions_controller_spec.rb
  require &#39;spec_helper&#39;

  describe "SessionsController" do
    describe "GET :new" do
      it "load the login page" do
      end
    end

    describe "POST :create" do
      it "stay on page if user is not found"
      it "stay on login page if user is not confirmed"
      it "stay on login page if user has wrong email"
      it "stay on login page if user has wrong password"
      it "redirect if user is correct"
    end

    describe "GET :logout" do
      it "empty the current session"
      it "redirect to homepage if user is logging out"
    end
  end
```

I>## Test-First development I> I> Is a term from Extreme Programming (XP) and means that you first I> write down your tests before writing any code to solve it. This forces you to really think about what you are I> going to do. These tests prevent you from over engineering a problem because you has to make these tests green.

Here are now the tests for the `GET :new` and `POST :create` actions of our session controller:

```
  # spec/app/controllers/sessions_controller_spec.rb

  require &#39;spec_helper&#39;
```

```
describe "SessionsController" do

  describe "GET :new" do
    it "load the login page" do
      get "/login"
      last_response.should be_ok
    end
  end

  describe "POST :create" do
    let(:user) { build(:user)}
    let(:params) { attributes_for(:user)}

    it "stay on page if user is not found" do
      User.should_receive(:find_by_email).and_return(false)
      post_create(user.attributes)
      last_response.should be_ok
    end

    it "stay on login page if user is not confirmed" do
      user.confirmation = false
      User.should_receive(:find_by_email).and_return(user)
      post_create(user.attributes)
      last_response.should be_ok
    end

    it "stay on login page if user has wrong email" do
      user.email = "fake@google.de"
      User.should_receive(:find_by_email).and_return(user)
      post_create(user.attributes)
      last_response.should be_ok
    end

    it "stay on login page if user has wrong password" do
      user.password = "test"
      User.should_receive(:find_by_email).and_return(user)
      post_create(user.attributes)
      last_response.should be_ok
    end

    it "redirect if user is correct" do
      user.confirmation = true
      User.should_receive(:find_by_email).and_return(user)
      post_create(user.attributes)
      last_response.should be_redirect
```

```
      end
    end

    private
    def post_create(params)
      post "sessions/create", params
    end

  end
```

We are using **mocking** to make test what we want with the `User.should_recei` method. I was thinking at the first that mocking is something very difficult but it isn't Read it the method out loud ten times and you can guess whats going on. If our `User` object gets call from it's class method `find_by_email` it should return our user object. This method will simulate from calling an actual find method in our application - yeah we are mocking the actual call and preventing our tests from hitting the database and making it faster. Actual call and preventing our tests from hitting the database and making it faster.

Here is the code for our session controller to make the test green:

```
  # app/controllers/session.rb

  JobVacancy::App.controllers :sessions do

    get :new, :map => "/login" do
      render &#39;sessions/new&#39;
    end

    post :create do
      user = User.find_by_email(params[:email])

      if user && user.confirmation && user.password == params[:password]
        sign_in(user)
        redirect &#39;/&#39;
      else
        render &#39;/sessions/new&#39;
      end
    end
```

```
    get :destroy, :map => &#39;/logout&#39; do
    end

  end
```

When I started the tests I got some weird error messages of calling a method on a nil object and spend one hour till I found the issue. Do you remember the `UserObserver`? Exactly, this tiny piece of code is also activated for our tests and since we disable sending mails with the `set :delive` settings in `app.rb` I never received an mails. The simple to this problem was to add an option to in the `spec_helper.rb` to disable the observer:

```
# spec/spec_helper.rb
...
RSpec.configure do |conf|
  conf.before do
    User.observers.disable :all # <-- turn of user observers for testing reasons,
  end
  ...
end
```

Running our tests:

```
$ rspec spec/app/controllers/sessions_controller_spec.rb

SessionsController
  GET :new
    load the login page
  POST :create
    stay on page if user is not found
    stay on login page if user is not confirmed
    stay on login page if user has wrong email
    stay on login page if user has wrong password
    redirect if user is correct
  GET :logout
    empty the current session (PENDING: Not yet implemented)
    redirect to homepage if user is logging out (PENDING: Not yet implemented)

Pending:
  SessionsController GET :logout empty the current session
    # Not yet implemented
    # ./spec/app/controllers/sessions_controller_spec.rb:52
  SessionsController GET :logout redirect to homepage if user is logging out
```

```
      # Not yet implemented
      # ./spec/app/controllers/sessions_controller_spec.rb:53

Finished in 0.62495 seconds
8 examples, 0 failures, 2 pending
```

Before going on with implementing the logout action we need to think what happened after we login. We have to find a mechanism to enable the information of the logged in user in all our controllers and views. Of course, we will do it with sessions. When we created the session controller there was the line `create  app/helpers/sessions_helper.rb` -- let's look into this file:

```
# app/helpers/sessions_helper.rb

# Helper methods defined here can be accessed in any controller or view in the ap

JobVacancy::App.helpers do
  # def simple_helper_method
  #  ...
  # end
end
```

Yeah, Padrino is so friendly to print the purpose of this new file and it basically says what we want to do. Let's implement the main features:

```
# app/helpers/session_helper.rb

JobVacancy::App.helpers do
  def current_user=(user)
    @current_user = user
  end

  def current_user
    @current_user ||= User.find_by_id(session[:current_user])
  end

  def sign_in(user)
    session[:current_user] = user.id
    self.current_user = user
  end
```

```
    def sign_out
      session.delete(:current_user)
    end

    def signed_in?
      !current_user.nil?
    end
  end
```

There's a lot of stuff going on in this helper:

- `current_user`: Uses the `||=` notation. If the left hand-side isn't initialized, initialize the left hand-side with the right hand-side.
- `sign_in(user)`: Uses the global session method use the user Id as login information
- `sign_out`: Purges the `:current_user` field from our session.
- `signed_in?`: We will use this small method within our whole application to display special actions which should only be available for authenticated users.

I>## Why Sessions and how does sign_out work? I> I> When you request an URL in your browser you are using the HTTP/HTTPS protocol. This protocol is stateless that means I> that it doesn't save the state in which you are in your application. Web applications implement states with one of I> the following mechanisms: hidden variables in forms when sending data, cookies, or query strings (e.g. I> http://localhost:3000/login?user=test&password=test). I> I> We are going to use cookies to save if a user is logged in and saving the user-Id in our session cookies under the I> `:current_user` key. I> I> What the delete method does is the following: It will look into the last request in your application inside the I> session information hash and delete the `current_user` key. And the sentence in code I> `browser.last_request`. If you want to explore more of the internal of an I> application I highly recommend you the Pry. You can throw in at any part of your I> application `binding.pry` and have full access to all variables.

Now we are in a position to write tests for our `:destroy` action:

```
# spec/app/controller/sessions_spec.rb

require &#39;spec_helper&#39;

describe "SessionsController" do
  ...
  describe "GET :logout" do
    it "empty the current session" do
      get_logout
      session[:current_user].should == nil
      last_response.should be_redirect
    end

    it "redirect to homepage if user is logging out" do
      get_logout
      last_response.should be_redirect
    end
  end

  private
  ...

  def get_logout
      # first arguments are params (like the ones out of an form), the second are
    get &#39;/logout&#39;, { :name => &#39;Hans&#39;, :password => &#39;Test123&#
  end
```

We use the our own `session` method in our tests to have access to the last response of our `rack.session`. What we need to achieve is to have access to Rack's SessionHash. The definition of this method is part of our `spec_helper.rb` method:

```
# spec/spec_helper.rb

...
# have access to the session variables
def session
  last_request.env[&#39;rack.session&#39;]
end
```

And finally the implementation of the code that it make our tests

green:

```
# app/controllers/session.rb

JobVacancy::App.controllers :sessions do
  get :destroy, :map => &#39;/logout&#39; do
    sign_out
    redirect &#39;/&#39;
  end
end
```

What we forget due to this point is to make use of the `sign_in(user)` method. Of course we need use this during our session `:create` action:

```
# app/controller/session.rb

JobVacancy::App.controllers :sessions do
  ...
  post :create do
    ...
    if user && user.confirmation && user.password == params[:password]
      sign_in(user)
      redirect &#39;/&#39;
    else
      ...
    end
  end

end
```

Where can we test now our logic? The main application layout of our application should have a "Login" and "Logout" link according to the status of the user:

```
# app/views/application.rb

<!DOCTYPE html>
<html lang="en-US">
  <%= stylesheet_link_tag &#39;../assets/application&#39; %>
  <%= javascript_include_tag &#39;../assets/application&#39; %>
</head>
<body>
  <div class=="container">
```

```
      <div class="row">
          <nav id="navigation">
          ...
          <% if signed_in? %>
            <%= link_to &#39;Logout&#39;, url_for(:sessions, :destroy) %>
          <% else %>
          <div class="span2">
            <%= link_to &#39;Login&#39;, url_for(:sessions, :new) %>
          </div>
          <% end %>
          </nav>
        </div>
        ...
      </div>
    </div>
  </body>
```

With the change above we changed the default "Registration" entry in our header navigation to "Login". We will add the link to the registration form now in the 'session/new' view:

```
  # app/views/sessions/new.erb

  <h1>Login</h1>

  <% form_tag &#39;/sessions/create&#39; do %>

    <%= label_tag :email %>
    <%= text_field_tag :email %>

    <%= label_tag :password %>
    <%= password_field_tag :password %>
    <p>
    <%= submit_tag "Sign up", :class => "btn btn-primary" %>
    </p>
  <% end %>

  New on this platform? <%= link_to &#39;Register&#39;, url_for(:users, :new) %>
```

Here we are using the form_tag instead of the `form_for` tag because we don't want to render information about a certain model. We want to use the information of the session form to find a user in our database. So we can use the submitted inputs with `params[:email]` and `params[:password]`

in the `:create` action in our action controller. My basic idea is to pass a variable to the rendering of method which says if we have an error or not and display the message accordingly. To handle this we are using the `:locals` option to create customized params for your views:

```
# app/controllers/sessions.rb

JobVacancy::App.controllers :sessions do

  get :new, :map => "/login" do
    render &#39;/sessions/new&#39;, :locals => { :error => false }
  end

  post :create do
    user = User.find_by_email(params[:email])

    if user && user.confirmation && user.password == params[:password]
      sign_in(user)
      redirect &#39;/&#39;
    else
      render &#39;/sessions/new&#39;, :locals => { :error => true }
    end
  end
  ...

end
```

Now we can simply use the `error` variable in our view:

```
# app/views/sessions/new.erb

<h1>Login</h1>

<% form_tag &#39;/sessions/create&#39; do %>
  <% if error %>
    <div class="alert alert-error">
      <h4>Error</h4>
      Your Email and/or Password is wrong!
    </div>
  <% end %>
...
```

```
  <% end %>

  New on this platform? <%= link_to &#39;Register&#39;, url_for(:users, :new) %>
```

The last thing we want to is to give the user feedback about what the action he was recently doing. Like that it would be nice to give feedback of the success of the logged and logged out action. We can do this with short flash messages above our application which will fade away after a certain amount of time. To do this we can use Padrino's flash mechanism is build on Rails flash message implementation.

And here is the implementation of the code:

```
# app/views/application.erb

<!DOCTYPE html>
<html lang="en-US">
<head>
  <title>Job Vacancy - find the best jobs</title>
  <%= stylesheet_link_tag &#39;../assets/application&#39; %>
  <%= javascript_include_tag &#39;../assets/application&#39; %>
</head>
<body>
  <div class="container">
    <% if flash[:notice] %>
      <div class="row" id="flash">
        <div class="span9 offset3 alert alert-success">
          <%= flash[:notice] %></p>
        </div>
      </div>
    <% end %>
  </div>
</body>
```

Next we need implement the flash messages in our session controller:

```
# app/controllers/sessions.rb

JobVacancy::App.controllers :sessions do
  ...
  post :create do
    user = User.find_by_email(params[:email])

    if user && user.confirmation && user.password == params[:password]
      flash[:notice] = "You have successfully logged out."
      sign_in(user)
      redirect &#39;/&#39;
    else
      render &#39;/sessions/new&#39;, :locals => { :error => true }
    end
  end
  ...
end
```

If you now login successfully you will see the message but it will stay there forever. But we don't want to have this message displayed the whole time, so we will use jQuery's fadeOut method to get rid of the message. Since we are first writing our own customized JavaScript, let's create the file with the following content:

```
# app/views/application.erb


<!DOCTYPE html>
<html lang="en-US">
<head>
  <title>Job Vacancy - find the best jobs</title>
  <%= stylesheet_link_tag &#39;../assets/application&#39; %>
  <%= javascript_include_tag &#39;../assets/application&#39; %>
</head>
<body>
  <div class=="container">
    <% if flash[:notice] %>
```

```
          <div class="row" id="flash">
            <div class="span9 offset3 alert alert-success">
              <%= flash[:notice] %></p>
            </div>
            <script type="text/javascript">
              $(function(){
                  $("#flash").fadeOut(2000);
              });
            </script>
          </div>
        <% end %>
      </div>
    </body>
```

Feel free to add the `flash[:notice]` function when the user has registered and confirmed successfully on our platform. If you have problems you can check my commit.

---

1. 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.

2. Got the inspiration from stackoverflow