



[Home](#) | [Book](#) | [Help](#) | [Contact](#) | [News](#)  | [Follow](#) 

Please see the post [Do not buy the print version of the Ruby on Rails Tutorial \(yet\)](#)

[skip to content](#) | [view as single page](#)

Ruby on Rails Tutorial

Learn Web Development with Rails

Michael Hartl

Contents

Chapter 1 From zero to deploy

1.1 Introduction

1.1.1 Comments for various readers

- 1.1.2 “Scaling” Rails
 - 1.1.3 Conventions in this book
- 1.2 Up and running
 - 1.2.1 Development environments
 - IDEs
 - Text editors and command lines
 - Browsers
 - A note about tools
 - 1.2.2 Ruby, RubyGems, Rails, and Git
 - Rails Installer (Windows)
 - Install Git
 - Install Ruby
 - Install RubyGems
 - Install Rails
 - 1.2.3 The first application
 - 1.2.4 Bundler
 - 1.2.5 rails server
 - 1.2.6 Model-view-controller (MVC)
- 1.3 Version control with Git
 - 1.3.1 Installation and setup
 - First-time system setup
 - First-time repository setup
 - 1.3.2 Adding and committing
 - 1.3.3 What good does Git do you?
 - 1.3.4 GitHub
 - 1.3.5 Branch, edit, commit, merge
 - Branch
 - Edit
 - Commit
 - Merge
 - Push
- 1.4 Deploying
 - 1.4.1 Heroku setup
 - 1.4.2 Heroku deployment, step one
 - 1.4.3 Heroku deployment, step two
 - 1.4.4 Heroku commands
- 1.5 Conclusion

Chapter 2 A demo app

- 2.1 Planning the application
 - 2.1.1 Modeling demo users
 - 2.1.2 Modeling demo microposts
- 2.2 The Users resource
 - 2.2.1 A user tour
 - 2.2.2 MVC in action
 - 2.2.3 Weaknesses of this Users resource
- 2.3 The Microposts resource
 - 2.3.1 A micropost microtour
 - 2.3.2 Putting the *micro* in microposts
 - 2.3.3 A user has_many microposts
 - 2.3.4 Inheritance hierarchies
 - 2.3.5 Deploying the demo app
- 2.4 Conclusion

Chapter 3 Mostly static pages

- 3.1 Static pages
 - 3.1.1 Truly static pages
 - 3.1.2 Static pages with Rails
- 3.2 Our first tests
 - 3.2.1 Test-driven development
 - 3.2.2 Adding a page
 - Red
 - Green
 - Refactor
- 3.3 Slightly dynamic pages
 - 3.3.1 Testing a title change
 - 3.3.2 Passing title tests
 - 3.3.3 Embedded Ruby
 - 3.3.4 Eliminating duplication with layouts
- 3.4 Conclusion
- 3.5 Exercises
- 3.6 Advanced setup
 - 3.6.1 Eliminating bundle exec
 - RVM Bundler integration
 - binstubs

- 3.6.2 Automated tests with Guard
- 3.6.3 Speeding up tests with Spork
Guard with Spork
- 3.6.4 Tests inside Sublime Text

Chapter 4 Rails-flavored Ruby

- 4.1 Motivation
- 4.2 Strings and methods
 - 4.2.1 Comments
 - 4.2.2 Strings
Printing
Single-quoted strings
 - 4.2.3 Objects and message passing
 - 4.2.4 Method definitions
 - 4.2.5 Back to the title helper
- 4.3 Other data structures
 - 4.3.1 Arrays and ranges
 - 4.3.2 Blocks
 - 4.3.3 Hashes and symbols
 - 4.3.4 CSS revisited
- 4.4 Ruby classes
 - 4.4.1 Constructors
 - 4.4.2 Class inheritance
 - 4.4.3 Modifying built-in classes
 - 4.4.4 A controller class
 - 4.4.5 A user class
- 4.5 Conclusion
- 4.6 Exercises

Chapter 5 Filling in the layout

- 5.1 Adding some structure
 - 5.1.1 Site navigation
 - 5.1.2 Bootstrap and custom CSS
 - 5.1.3 Partials
- 5.2 Sass and the asset pipeline
 - 5.2.1 The asset pipeline
Asset directories

- Manifest files
 - Preprocessor engines
 - Efficiency in production
- 5.2.2 Syntactically awesome stylesheets
 - Nesting
 - Variables
- 5.3 Layout links
 - 5.3.1 Route tests
 - 5.3.2 Rails routes
 - 5.3.3 Named routes
 - 5.3.4 Pretty RSpec
- 5.4 User signup: A first step
 - 5.4.1 Users controller
 - 5.4.2 Signup URI
- 5.5 Conclusion
- 5.6 Exercises

Chapter 6 Modeling users

- 6.1 User model
 - 6.1.1 Database migrations
 - 6.1.2 The model file
 - Model annotation
 - Accessible attributes
 - 6.1.3 Creating user objects
 - 6.1.4 Finding user objects
 - 6.1.5 Updating user objects
- 6.2 User validations
 - 6.2.1 Initial user tests
 - 6.2.2 Validating presence
 - 6.2.3 Length validation
 - 6.2.4 Format validation
 - 6.2.5 Uniqueness validation
 - The uniqueness caveat
- 6.3 Adding a secure password
 - 6.3.1 An encrypted password
 - 6.3.2 Password and confirmation
 - 6.3.3 User authentication

- 6.3.4 User has secure password
 - 6.3.5 Creating a user
- 6.4 Conclusion
- 6.5 Exercises

Chapter 7 Sign up

- 7.1 Showing users
 - 7.1.1 Debug and Rails environments
 - 7.1.2 A Users resource
 - 7.1.3 Testing the user show page (with factories)
 - 7.1.4 A Gravatar image and a sidebar
- 7.2 Signup form
 - 7.2.1 Tests for user signup
 - 7.2.2 Using `form_for`
 - 7.2.3 The form HTML
- 7.3 Signup failure
 - 7.3.1 A working form
 - 7.3.2 Signup error messages
- 7.4 Signup success
 - 7.4.1 The finished signup form
 - 7.4.2 The flash
 - 7.4.3 The first signup
 - 7.4.4 Deploying to production with SSL
- 7.5 Conclusion
- 7.6 Exercises

Chapter 8 Sign in, sign out

- 8.1 Sessions and signin failure
 - 8.1.1 Sessions controller
 - 8.1.2 Signin tests
 - 8.1.3 Signin form
 - 8.1.4 Reviewing form submission
 - 8.1.5 Rendering with a flash message
- 8.2 Signin success
 - 8.2.1 Remember me
 - 8.2.2 A working `sign_in` method
 - 8.2.3 Current user

- 8.2.4 Changing the layout links
 - 8.2.5 Signin upon signup
 - 8.2.6 Signing out
- 8.3 Introduction to Cucumber (optional)
 - 8.3.1 Installation and setup
 - 8.3.2 Features and steps
 - 8.3.3 Counterpoint: RSpec custom matchers
- 8.4 Conclusion
- 8.5 Exercises

Chapter 9 Updating, showing, and deleting users

- 9.1 Updating users
 - 9.1.1 Edit form
 - 9.1.2 Unsuccessful edits
 - 9.1.3 Successful edits
- 9.2 Authorization
 - 9.2.1 Requiring signed-in users
 - 9.2.2 Requiring the right user
 - 9.2.3 Friendly forwarding
- 9.3 Showing all users
 - 9.3.1 User index
 - 9.3.2 Sample users
 - 9.3.3 Pagination
 - 9.3.4 Partial refactoring
- 9.4 Deleting users
 - 9.4.1 Administrative users
 - Revisiting `attr_accessible`
 - 9.4.2 The destroy action
- 9.5 Conclusion
- 9.6 Exercises

Chapter 10 User microposts

- 10.1 A Micropost model
 - 10.1.1 The basic model
 - 10.1.2 Accessible attributes and the first validation
 - 10.1.3 User/Micropost associations
 - 10.1.4 Micropost refinements

- Default scope
 - Dependent: destroy
 - 10.1.5 Content validations
- 10.2 Showing microposts
 - 10.2.1 Augmenting the user show page
 - 10.2.2 Sample microposts
- 10.3 Manipulating microposts
 - 10.3.1 Access control
 - 10.3.2 Creating microposts
 - 10.3.3 A proto-feed
 - 10.3.4 Destroying microposts
- 10.4 Conclusion
- 10.5 Exercises

Chapter 11 Following users

- 11.1 The Relationship model
 - 11.1.1 A problem with the data model (and a solution)
 - 11.1.2 User/relationship associations
 - 11.1.3 Validations
 - 11.1.4 Followed users
 - 11.1.5 Followers
- 11.2 A web interface for following users
 - 11.2.1 Sample following data
 - 11.2.2 Stats and a follow form
 - 11.2.3 Following and followers pages
 - 11.2.4 A working follow button the standard way
 - 11.2.5 A working follow button with Ajax
- 11.3 The status feed
 - 11.3.1 Motivation and strategy
 - 11.3.2 A first feed implementation
 - 11.3.3 Subselects
 - 11.3.4 The new status feed
- 11.4 Conclusion
 - 11.4.1 Extensions to the sample application
 - Replies
 - Messaging
 - Follower notifications

Foreword

My former company (CD Baby) was one of the first to loudly switch to Ruby on Rails, and then even more loudly switch back to PHP (Google me to read about the drama). This book by Michael Hartl came so highly recommended that I had to try it, and the *Ruby on Rails Tutorial* is what I used to switch back to Rails again.

Though I've worked my way through many Rails books, this is the one that finally made me “get” it. Everything is done very much “the Rails way”—a way that felt very unnatural to me before, but now after doing this book finally feels natural. This is also the only Rails book that does test-driven development the entire time, an approach highly recommended by the experts but which has never been so clearly demonstrated before. Finally, by including Git, GitHub, and Heroku in the demo examples, the author really gives you a feel for what it's like to do a real-world project. The tutorial's code examples are not in isolation.

The linear narrative is such a great format. Personally, I powered through the *Rails Tutorial* in three long days, doing all the examples and challenges at the end of each chapter. Do it from start to finish, without jumping around, and you'll get the ultimate benefit.

Enjoy!

[Derek Sivers \(sivers.org\)](http://sivers.org)

Formerly: Founder, [CD Baby](#)

Currently: Founder, [Thoughts Ltd.](#)

Acknowledgments

The *Ruby on Rails Tutorial* owes a lot to my previous Rails book, *RailsSpace*, and hence to my coauthor [Aurelius Prochazka](#). I'd like to thank Aure both for the work he did on that book and for his support of this one. I'd also like to thank Debra Williams Cauley, my editor on both *RailsSpace* and the *Ruby on Rails Tutorial*; as long as she keeps taking me to baseball games, I'll keep writing books for her.

I'd like to acknowledge a long list of Rubyists who have taught and inspired me over the years: David Heinemeier Hansson, Yehuda Katz, Carl Lerche, Jeremy Kemper, Xavier Noria, Ryan Bates, Geoffrey Grosenbach, Peter Cooper, Matt Aimonetti, Gregg Pollack, Wayne E. Seguin, Amy Hoy, Dave Chelimsky, Pat Maddox, Tom Preston-Werner, Chris Wanstrath, Chad Fowler, Josh Susser, Obie Fernandez, Ian McFarland, Steven Bristol, Wolfram Arnold, Alex Chaffee, Giles Bowkett, Evan Dorn, Long Nguyen, James Lindenbaum, Adam Wiggins, Tikhon Bernstam, Ron Evans, Wyatt Greene, Miles Forrest, the good people at Pivotal Labs, the Heroku gang, the thoughtbot guys, and the GitHub crew. Finally, many, many readers—far too many to list—have contributed a huge number of bug reports and suggestions during the writing of this book, and I gratefully acknowledge their help in making it as good as it can be.

About the author

[Michael Hartl](#) is the author of the [Ruby on Rails Tutorial](#), the leading introduction to web development with [Ruby on Rails](#). His prior experience includes writing and developing *RailsSpace*, an extremely obsolete Rails tutorial book, and developing Insoshi, a once-popular and now-obsolete social networking platform in Ruby on Rails. In 2011, Michael received a [Ruby Hero Award](#) for his contributions to the Ruby community. He is a graduate of [Harvard College](#), has a [Ph.D. in Physics](#) from [Caltech](#), and is an alumnus of the [Y Combinator](#) entrepreneur program.

Copyright and license

Ruby on Rails Tutorial: Learn Web Development with Rails. Copyright © 2012 by Michael Hartl.

All source code in the *Ruby on Rails Tutorial* is available jointly under the [MIT License](#) and the [Beerware License](#).

The MIT License

Copyright (c) 2012 Michael Hartl

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

```
/*
 * -----
 * "THE BEER-WARE LICENSE" (Revision 42):
 * Michael Hartl wrote this code. As long as you retain this notice you
 * can do whatever you want with this stuff. If we meet some day, and you think
 * this stuff is worth it, you can buy me a beer in return.
 * -----
 */
```

Chapter 3

Mostly static pages

In this chapter, we will begin developing the sample application that will serve as our example throughout the rest of this tutorial. Although the sample app will eventually have users, microposts, and a full login and authentication framework, we will begin with a seemingly limited topic: the creation of static pages. Despite its apparent simplicity, making static pages is a highly instructive exercise, rich in implications—a perfect start for our nascent application.

Although Rails is designed for making database-backed dynamic websites, it also excels at making the kind of static pages we might make with raw HTML files. In fact, using Rails even for static pages yields a distinct advantage: we can easily add just a *small* amount of dynamic content. In this chapter we'll learn how. Along the way, we'll get our first taste of *automated testing*, which will help us be more confident that our code is correct. Moreover, having a good test suite will allow us to *refactor* our code with confidence, changing its form without changing its function.

There's a lot of code in this chapter, especially in [Section 3.2](#) and [Section 3.3](#), and if you're new to Ruby you shouldn't worry about understanding the details right now. As noted in [Section 1.1.1](#), one strategy is to copy-and-paste the tests and use them to verify the application code, without worrying at this point how they work. In addition, [Chapter 4](#) covers Ruby in more depth, so there is plenty of opportunity for these ideas to sink in. Finally, RSpec tests will recur throughout the tutorial, so if you get stuck now I recommend forging ahead; you'll be amazed how, after just a few more chapters, initially inscrutable code will suddenly look simple.

As in [Chapter 2](#), before getting started we need to create a new Rails project, this time called



sample_app:

```
$ cd ~/rails_projects
$ rails new sample_app --skip-test-unit
$ cd sample_app
```

Here the `--skip-test-unit` option to the `rails` command tells Rails not to generate a `test` directory associated with the default `Test::Unit` framework. This is not because we won't be writing tests; on the contrary, starting in [Section 3.2](#) we will be using an alternate testing framework called *RSpec* to write a thorough test suite.

As in [Section 2.1](#), our next step is to use a text editor to update the `Gemfile` with the gems needed by our application. On the other hand, for the sample application we'll also need two gems we didn't need before: the gem for RSpec and the gem for the RSpec library specific to Rails. The code to include them is shown in [Listing 3.1](#). (Note: If you would like to install *all* the gems needed for the sample application, you should use the code in [Listing 9.49](#) at this time.)

Listing 3.1. A `Gemfile` for the sample app.

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

gem 'rails', '3.2.8'

group :development, :test do
  gem 'sqlite3', '1.3.5'
  gem 'rspec-rails', '2.11.0'
end

# Gems used only for assets and not required
# in production environments by default.
group :assets do
  gem 'sass-rails', '3.2.5'
  gem 'coffee-rails', '3.2.2'
  gem 'uglifier', '1.2.3'
```

```
end

gem 'jquery-rails', '2.0.2'

group :test do
  gem 'capybara', '1.1.2'
end

group :production do
  gem 'pg', '0.12.2'
end
```

This includes `rspec-rails` in development mode so that we have access to RSpec-specific generators, and it includes it in test mode in order to run the tests. We don't have to install RSpec itself because it is a dependency of `rspec-rails` and will thus be installed automatically. We also include the [Capybara gem](#), which allows us to simulate a user's interaction with the sample application using a natural English-like syntax.¹ As in [Chapter 2](#), we also must include the PostgreSQL gem in production for deployment to Heroku:

```
group :production do
  gem 'pg', '0.12.2'
end
```

Heroku recommends against using different databases in development and production, but for the sample application it won't make any difference, and SQLite is *much* easier than PostgreSQL to install and configure. Installing and configuring PostgreSQL on your local machine is left as an exercise ([Section 3.5](#)).

To install and include the new gems, we run **bundle install**:

```
$ bundle install --without production
```

As in [Chapter 2](#), we suppress the installation of production gems using the option `--without production`. This is a “remembered option”, which means that we don’t have to include it in future invocations of Bundler. Instead, we can write simply `bundle install`.²

Next, we need to configure Rails to use RSpec in place of `Test::Unit`. This can be accomplished with `rails generate rspec:install`:

```
$ rails generate rspec:install
```

If your system complains about the lack of a JavaScript runtime, visit the [execjs page at GitHub](#) for a list of possibilities. I particularly recommend installing [Node.js](#).

With that, all we have left is to initialize the Git repository:³

```
$ git init
$ git add .
$ git commit -m "Initial commit"
```

As with the first application, I suggest updating the `README` file (located in the root directory of the application) to be more helpful and descriptive, as shown in [Listing 3.2](#).

Listing 3.2. An improved `README` file for the sample app.

```
# Ruby on Rails Tutorial: sample application

This is the sample application for
[*Ruby on Rails Tutorial: Learn Rails by Example*](http://railstutorial.org/)
by [Michael Hartl](http://michaelhartl.com/).
```

Then change it to use the Markdown extension `.md` and commit the changes:

```
$ git mv README.rdoc README.md  
$ git commit -a -m "Improve the README"
```

Create a New Repository

GitHub, Inc. (US) <https://github.com/new>

github Search... Explore Gist Blog Help mhartl

Owner: mhartl Repository name: sample_app ✓

Great repository names are short and memorable. Need inspiration? How about [derp-octo-adventure](#).

Description (optional): Ruby on Rails Tutorial sample application

☒ Public 🌞
Anyone can see this repository. You choose who can commit.

☐ Private 🗑️
You choose who can see and commit to this repository.

☐ Initialize this repository with a README
This will allow you to `git clone` the repository immediately.

Add .gitignore: None

Create repository

GitHub Tools Extras Documentation

About Gauges: Analyze web traffic GitHub Shop GitHub Help
Blog Speaker Deck: Presentations The Octodex Developer API
Features Gist: Code snippets GitHub Flavored Markdown
Contact & Support GitHub for Mac GitHub Pages

Figure 3.1: Creating the sample app repository at GitHub. ([full size](#))

Since we'll be using this sample app throughout the rest of the book, it's a good idea to make a repository at GitHub ([Figure 3.1](#)) and push it up:

```
$ git remote add origin git@github.com:<username>/sample_app.git
$ git push -u origin master
```

As a result of my performing this step, you can find the [Rails Tutorial sample application code on GitHub](#) (under a slightly different name).⁴

Of course, we can optionally deploy the app to Heroku even at this early stage:

```
$ heroku create --stack cedar
$ git push heroku master
```

As you proceed through the rest of the book, I recommend pushing and deploying the application regularly:

```
$ git push
$ git push heroku
```

This provides remote backups and lets you catch any production errors as soon as possible. If you run into problems at Heroku, make sure to take a look at the production logs to try to diagnose the problem:

```
$ heroku logs
```

With all the preparation finished, we're finally ready to get started developing the sample application.

3.1 Static pages

Rails has two main ways of making static web pages. First, Rails can handle *truly* static pages consisting of raw HTML files. Second, Rails allows us to define *views* containing raw HTML, which Rails can *render* so that the web server can send it to the browser.

In order to get our bearings, it's helpful to recall the Rails directory structure from [Section 1.2.3](#) ([Figure 1.2](#)). In this section, we'll be working mainly in the **app/controllers** and **app/views** directories. (In [Section 3.2](#), we'll even add a new directory of our own.)

This is the first section where it's useful to be able to open the entire Rails directory in your text editor or IDE. Unfortunately, how to do this is system-dependent, but in many cases you can open the current application directory, represented in Unix by a dot `.`, using the command-line command for your editor of choice:

```
$ cd ~/rails_projects/sample_app  
$ <editor name> .
```

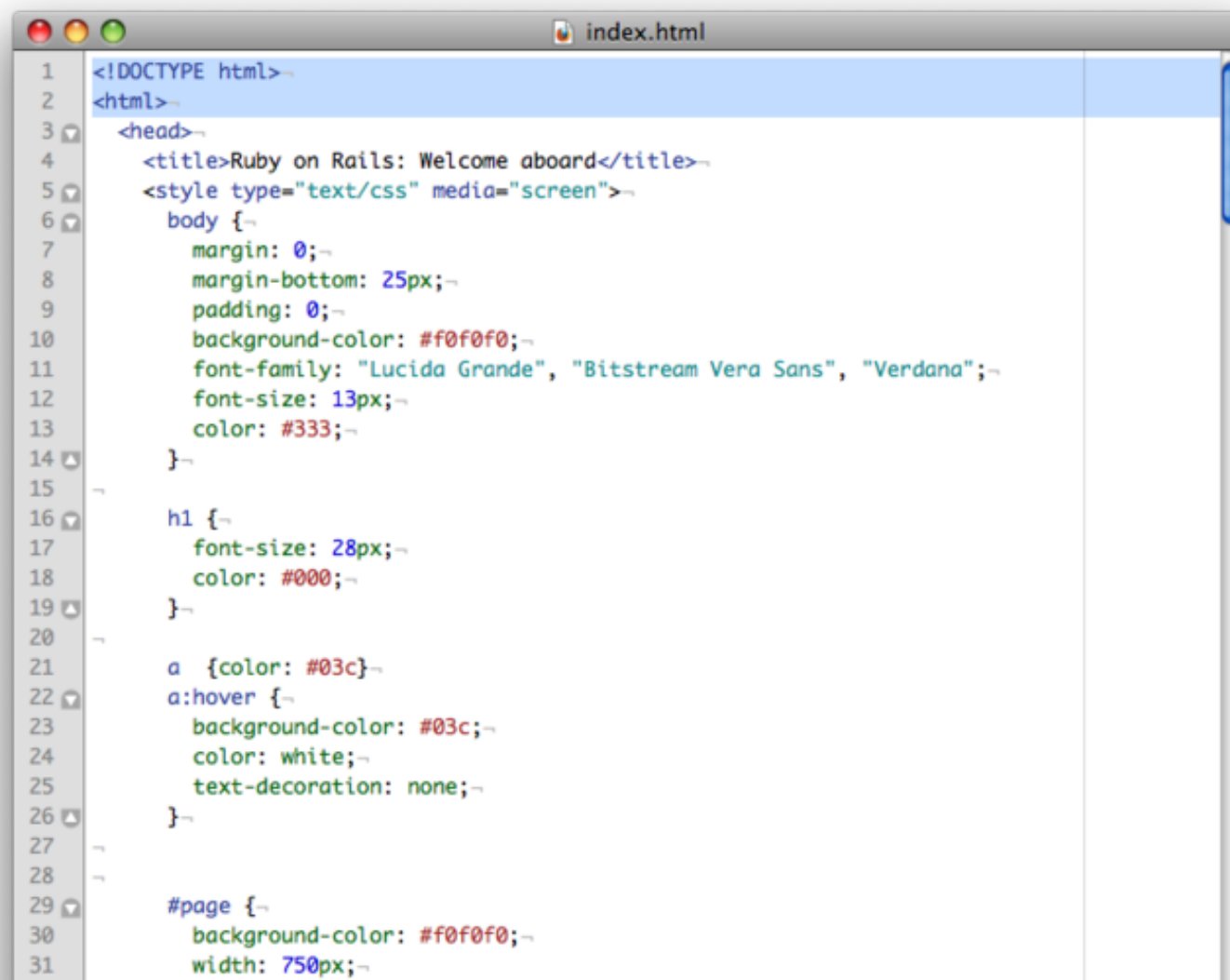
For example, to open the sample app in Sublime Text, you type

```
$ subl .
```

For Vim, you type `vim .`, `gvim .`, or `mvim .` depending on which flavor of Vim you use.

3.1.1 Truly static pages

We start with truly static pages. Recall from [Section 1.2.5](#) that every Rails application comes with a minimal working application thanks to the `rails` script, with a default welcome page at the address <http://localhost:3000/> ([Figure 1.3](#)).



```
1 <!DOCTYPE html>~
2 <html>~
3 <head>~
4 <title>Ruby on Rails: Welcome aboard</title>~
5 <style type="text/css" media="screen">~
6   body {~
7     margin: 0;~
8     margin-bottom: 25px;~
9     padding: 0;~
10    background-color: #f0f0f0;~
11    font-family: "Lucida Grande", "Bitstream Vera Sans", "Verdana";~
12    font-size: 13px;~
13    color: #333;~
14  }~
15  ~
16  h1 {~
17    font-size: 28px;~
18    color: #000;~
19  }~
20  ~
21  a {color: #03c;}~
22  a:hover {~
23    background-color: #03c;~
24    color: white;~
25    text-decoration: none;~
26  }~
27  ~
28  ~
29  #page {~
30    background-color: #f0f0f0;~
31    width: 750px;~
```

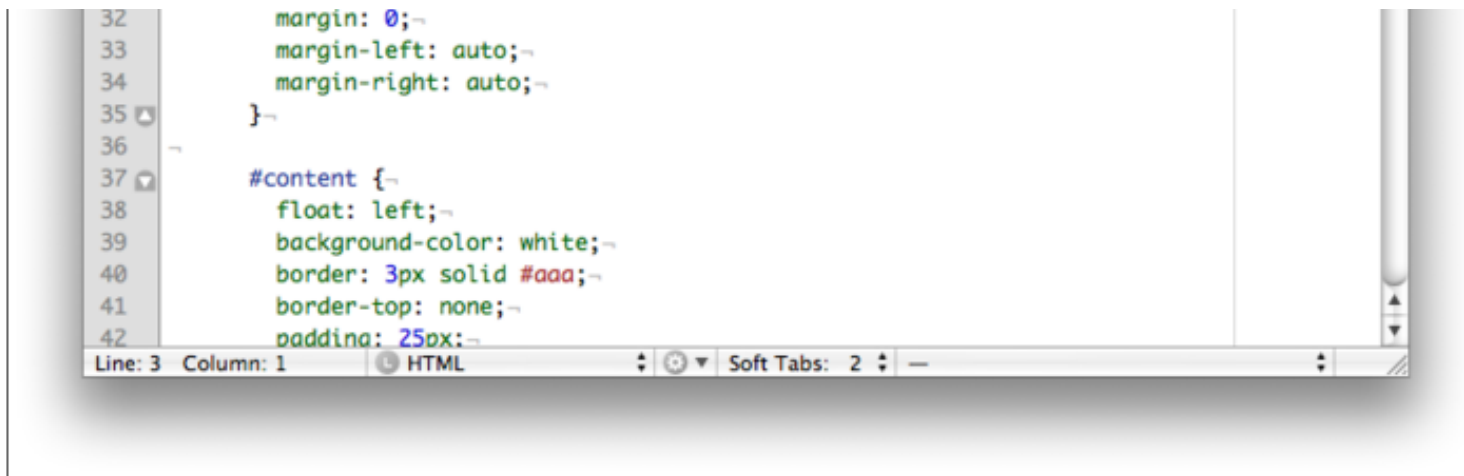


Figure 3.2: The `public/index.html` file. ([full size](#))

To learn where this page comes from, take a look at the file `public/index.html` ([Figure 3.2](#)). Because the file contains its own stylesheet information, it's a little messy, but it gets the job done: by default, Rails serves any files in the `public` directory directly to the browser.⁵ In the case of the special `index.html` file, you don't even have to indicate the file in the URI, as `index.html` is the default. You can include it if you want, though; the addresses `http://localhost:3000/` and `http://localhost:3000/index.html` are equivalent.

As you might expect, if we want we can make our own static HTML files and put them in the same `public` directory as `index.html`. For example, let's create a file with a friendly greeting ([Listing 3.3](#)):⁶

```
$ subl public/hello.html
```

Listing 3.3. A typical HTML file, with a friendly greeting.

`public/hello.html`

```
<!DOCTYPE html>
<html>
  <head>
    <title>Greeting</title>
  </head>
  <body>
    <p>Hello, world!</p>
  </body>
</html>
```

We see in [Listing 3.3](#) the typical structure of an HTML document: a *document type*, or doctype, declaration at the top to tell browsers which version of HTML we’re using (in this case, [HTML5](#));⁷ a **head** section, in this case with “Greeting” inside a **title** tag; and a **body** section, in this case with “Hello, world!” inside a **p** (paragraph) tag. (The indentation is optional—HTML is not sensitive to whitespace, and ignores both tabs and spaces—but it makes the document’s structure easier to see.)

Now run a local server using

```
$ rails server
```

and navigate to <http://localhost:3000/hello.html>. As promised, Rails renders the page straightaway ([Figure 3.3](#)). Note that the title displayed at the top of the browser window in [Figure 3.3](#) is just the contents inside the **title** tag, namely, “Greeting”.

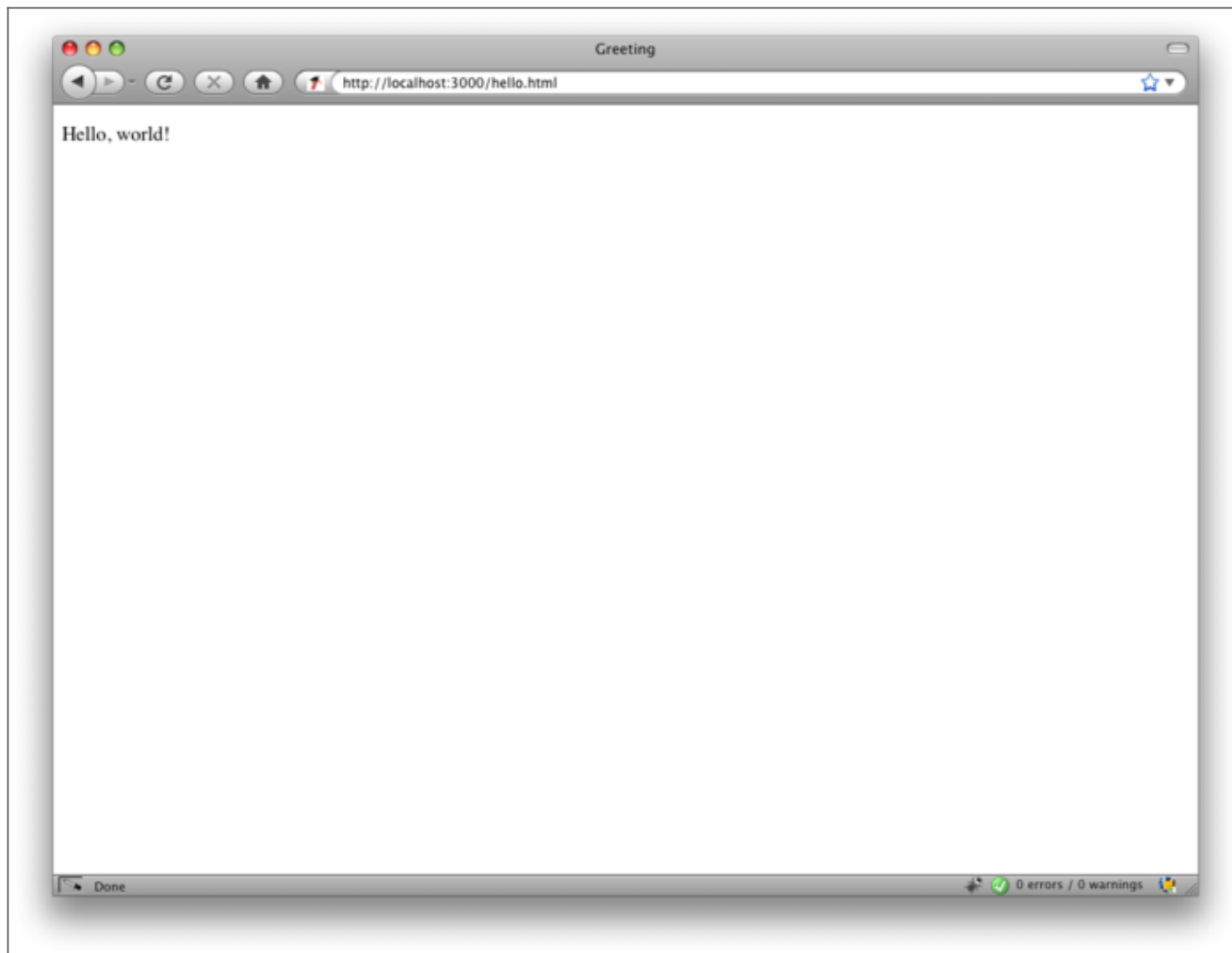


Figure 3.3: A new static HTML file. [\(full size\)](#)

Since this file is just for demonstration purposes, we don't really want it to be part of our sample application, so it's probably best to remove it once the thrill of creating it has worn off:

```
$ rm public/hello.html
```

We'll leave the `index.html` file alone for now, but of course eventually we should remove it: we don't want the root of our application to be the Rails default page shown in [Figure 1.3](#). We'll see in [Section 5.3](#) how to change the address <http://localhost:3000/> to point to something other than `public/index.html`.

3.1.2 Static pages with Rails

The ability to return static HTML files is nice, but it's not particularly useful for making dynamic web applications. In this section, we'll take a first step toward making dynamic pages by creating a set of Rails *actions*, which are a more powerful way to define URIs than static files.⁸ Rails actions come bundled together inside *controllers* (the C in MVC from [Section 1.2.6](#)), which contain sets of actions related by a common purpose. We got a glimpse of controllers in [Chapter 2](#), and will come to a deeper understanding once we explore the [REST architecture](#) more fully (starting in [Chapter 6](#)); in essence, a controller is a container for a group of (possibly dynamic) web pages.

To get started, recall from [Section 1.3.5](#) that, when using Git, it's a good practice to do our work on a separate topic branch rather than the master branch. If you're using Git for version control, you should run the following command:

```
$ git checkout -b static-pages
```

Rails comes with a script for making controllers called `generate`; all it needs to work its magic is the controller's name. In order to use `generate` with RSpec, you need to run the RSpec generator command if you didn't run it when following the introduction to this chapter:

```
$ rails generate rspec:install
```

Since we'll be making a controller to handle static pages, we'll call it the StaticPages controller. We'll also plan to make actions for a Home page, a Help page, and an About page. The **generate** script takes an optional list of actions, so we'll include two of the initial actions directly on the command line ([Listing 3.4](#)).

Listing 3.4. Generating a StaticPages controller.

```
$ rails generate controller StaticPages home help --no-test-framework
  create  app/controllers/static_pages_controller.rb
  route   get "static_pages/help"
  route   get "static_pages/home"
  invoke  erb
  create  app/views/static_pages
  create  app/views/static_pages/home.html.erb
  create  app/views/static_pages/help.html.erb
  invoke  helper
  create  app/helpers/static_pages_helper.rb
  invoke  assets
  invoke  coffee
  create  app/assets/javascripts/static_pages.js.coffee
  invoke  scss
  create  app/assets/stylesheets/static_pages.css.scss
```

Note that we've used the option `--no-test-framework` to suppress the generation of the default RSpec tests, which we won't be using. Instead, we'll create the tests by hand starting in [Section 3.2](#). We've also intentionally left off the **about** action from the command line arguments in [Listing 3.4](#) so that we can see how to add it using test-driven development, or TDD ([Section 3.2](#)).

By the way, if you ever make a mistake when generating code, it's useful to know how to reverse the process. See [Box 3.1](#) for some techniques on how to undo things in Rails.

Box 3.1. Undoing things

Even when you're very careful, things can sometimes go wrong when developing Rails applications. Happily, Rails has some facilities to help you recover.

One common scenario is wanting to undo code generation—for example, if you change your mind on the name of a controller. When generating a controller, Rails creates many more files than the controller file itself (as seen in [Listing 3.4](#)). Undoing the generation means removing not only the principal generated file, but all the ancillary files as well. (In fact, we also want to undo any automatic edits made to the `routes.rb` file.) In Rails, this can be accomplished with `rails destroy`. In particular, these two commands cancel each other out:

```
$ rails generate controller FooBars baz quux
$ rails destroy controller FooBars baz quux
```

Similarly, in [Chapter 6](#) we'll generate a *model* as follows:

```
$ rails generate model Foo bar:string baz:integer
```

This can be undone using

```
$ rails destroy model Foo
```

(In this case, it turns out we can omit the other command-line arguments. When you get to [Chapter 6](#), see if you can figure out why.)

Another technique related to models involves undoing *migrations*, which we saw briefly in [Chapter 2](#) and will see much more of starting in [Chapter 6](#). Migrations change the state of the database using

```
$ rake db:migrate
```

We can undo a single migration step using

```
$ rake db:rollback
```

To go all the way back to the beginning, we can use

```
$ rake db:migrate VERSION=0
```

As you might guess, substituting any other number for 0 migrates to that version number, where the version numbers come from listing the migrations sequentially.

With these techniques in hand, we are well-equipped to recover from the inevitable development [snafus](#).

The StaticPages controller generation in [Listing 3.4](#) automatically updates the *routes* file, called **config/routes.rb**, which Rails uses to find the correspondence between URIs and web pages. This is our first encounter with the **config** directory, so it's helpful to take a quick look at it ([Figure 3.4](#)). The **config** directory is where Rails collects files needed for the application configuration—hence the name.

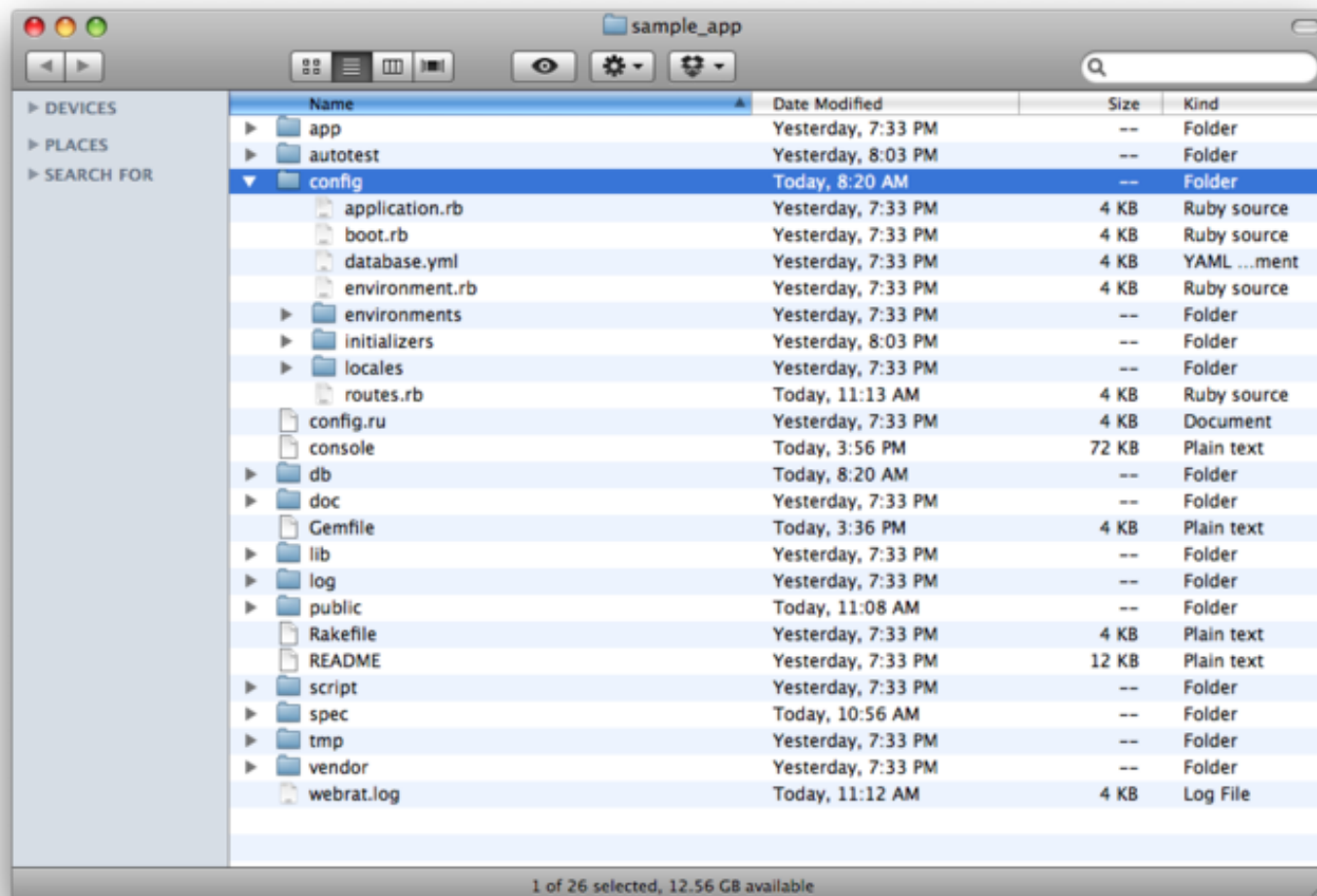


Figure 3.4: Contents of the sample app's **config** directory. ([full size](#))

Since we generated **home** and **help** actions, the routes file already has a rule for each one, as seen in [Listing 3.5](#).

Listing 3.5. The routes for the **home** and **help** actions in the StaticPages controller.

config/routes.rb

```
SampleApp::Application.routes.draw do
  get "static_pages/home"
  get "static_pages/help"
  .
  .
  .
end
```

Here the rule

```
get "static_pages/home"
```

maps requests for the URI /static_pages/home to the **home** action in the StaticPages controller. Moreover, by using **get** we arrange for the route to respond to a GET request, which is one of the fundamental *HTTP verbs* supported by the hypertext transfer protocol ([Box 3.2](#)). In our case, this means that when we generate a **home** action inside the StaticPages controller we automatically get a page at the address /static_pages/home. To see the results, navigate to [/static_pages/home](#) ([Figure 3.5](#)).

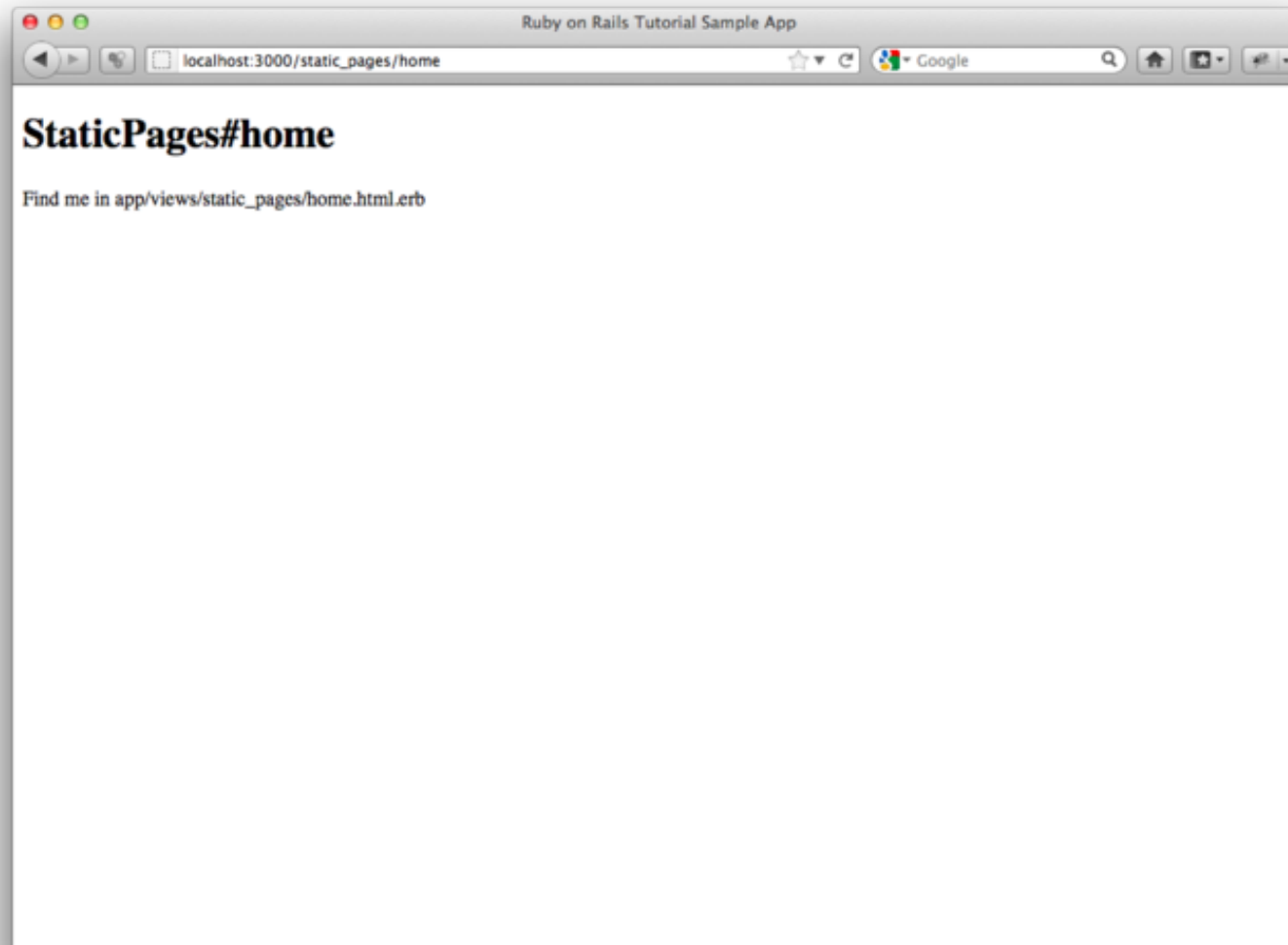


Figure 3.5: The raw home view (/static_pages/home). ([full size](#))

Box 3.2. GET, et cet.

The hypertext transfer protocol ([HTTP](#)) defines four basic operations, corresponding to the four verbs *get*, *post*, *put*, and *delete*. These refer to operations between a *client* computer

(typically running a web browser such as Firefox or Safari) and a *server* (typically running a web server such as Apache or Nginx). (It's important to understand that, when developing Rails applications on a local computer, the client and server are the same physical machine, but in general they are different.) An emphasis on HTTP verbs is typical of web frameworks (including Rails) influenced by the *REST architecture*, which we saw briefly in [Chapter 2](#) and will start learning about more in [Chapter 7](#).

GET is the most common HTTP operation, used for *reading* data on the web; it just means “get a page”, and every time you visit a site like google.com or wikipedia.org your browser is submitting a GET request. POST is the next most common operation; it is the request sent by your browser when you submit a form. In Rails applications, POST requests are typically used for *creating* things (although HTTP also allows POST to perform updates); for example, the POST request sent when you submit a registration form creates a new user on the remote site. The other two verbs, PUT and DELETE, are designed for *updating* and *destroying* things on the remote server. These requests are less common than GET and POST since browsers are incapable of sending them natively, but some web frameworks (including Ruby on Rails) have clever ways of making it *seem* like browsers are issuing such requests.

To understand where this page comes from, let's start by taking a look at the StaticPages controller in a text editor; you should see something like [Listing 3.6](#). You may note that, unlike the demo Users and Microposts controllers from [Chapter 2](#), the StaticPages controller does not use the standard REST actions. This is normal for a collection of static pages—the REST architecture isn't the best solution to every problem.

Listing 3.6. The StaticPages controller made by [Listing 3.4](#).

`app/controllers/static_pages_controller.rb`

```
class StaticPagesController < ApplicationController
```

```
def home
end

def help
end
end
```

We see from the `class` keyword in [Listing 3.6](#) that `static_pages_controller.rb` defines a *class*, in this case called `StaticPagesController`. Classes are simply a convenient way to organize *functions* (also called *methods*) like the `home` and `help` actions, which are defined using the `def` keyword. The angle bracket `<` indicates that `StaticPagesController` *inherits* from the Rails class `ApplicationController`; as we'll see momentarily, this means that our pages come equipped with a large amount of Rails-specific functionality. (We'll learn more about both classes and inheritance in [Section 4.4](#).)

In the case of the StaticPages controller, both its methods are initially empty:

```
def home
end

def help
end
```

In plain Ruby, these methods would simply do nothing. In Rails, the situation is different; `StaticPagesController` is a Ruby class, but because it inherits from `ApplicationController` the behavior of its methods is specific to Rails: when visiting the URI `/static_pages/home`, Rails looks in the StaticPages controller and executes the code in the `home` action, and then renders the *view* (the V in MVC from [Section 1.2.6](#)) corresponding to the action. In the present case, the `home` action is empty, so all visiting `/static_pages/home` does is render the view. So, what does a view look like, and how do we find it?

If you take another look at the output in [Listing 3.4](#), you might be able to guess the correspondence between actions and views: an action like **home** has a corresponding view called **home.html.erb**. We'll learn in [Section 3.3](#) what the **.erb** part means; from the **.html** part you probably won't be surprised that it basically looks like HTML ([Listing 3.7](#)).

Listing 3.7. The generated view for the Home page.

app/views/static_pages/home.html.erb

```
<h1>StaticPages#home</h1>
<p>Find me in app/views/static_pages/home.html.erb</p>
```

The view for the **help** action is analogous ([Listing 3.8](#)).

Listing 3.8. The generated view for the Help page.

app/views/static_pages/help.html.erb

```
<h1>StaticPages#help</h1>
<p>Find me in app/views/static_pages/help.html.erb</p>
```

Both of these views are just placeholders: they have a top-level heading (inside the **h1** tag) and a paragraph (**p** tag) with the full path to the relevant file. We'll add some (very slightly) dynamic content starting in [Section 3.3](#), but as they stand these views underscore an important point: Rails views can simply contain static HTML. As far as the browser is concerned, the raw HTML files from [Section 3.1.1](#) and the controller/action method of delivering pages are indistinguishable: all the browser ever sees is HTML.

In the remainder of this chapter, we'll add some custom content to the Home and Help pages, and then add in the About page we left off in [Section 3.1.2](#). Then we'll add a very small amount of dynamic content by changing the title on a per-page basis.

Before moving on, if you're using Git it's a good idea to add the files for the StaticPages controller to the repository:

```
$ git add .  
$ git commit -m "Add a StaticPages controller"
```

3.2 Our first tests

The *Rails Tutorial* takes an intuitive approach to testing that emphasizes the behavior of the application rather than its precise implementation, a variant of test-driven development (TDD) known as behavior-driven development (BDD). Our main tools will be *integration tests* (starting in this section) and *unit tests* (starting in [Chapter 6](#)). Integration tests, known as *request specs* in the context of RSpec, allow us to simulate the actions of a user interacting with our application using a web browser. Together with the natural-language syntax provided by Capybara, integration tests provide a powerful method to test our application's functionality without having to manually check each page with a browser. (Another popular choice for BDD, called Cucumber, is introduced in [Section 8.3](#).)

The defining quality of TDD is writing tests *first*, before the application code. Initially, this might take some getting used to, but the benefits are significant. By writing a *failing* test first and then implementing the application code to get it to pass, we increase our confidence that the test is actually covering the functionality we think it is. Moreover, the fail-implement-pass development cycle induces a [flow state](#), leading to enjoyable coding and high productivity. Finally, the tests act as a *client* for the application code, often leading to more elegant software designs.

It's important to understand that TDD is not always the right tool for the job: there's no reason to dogmatically insist that tests always should be written first, that they should cover every single feature, or that there should necessarily be any tests at all. For example, when you aren't at all sure

how to solve a given programming problem, it's often useful to skip the tests and write only application code, just to get a sense of what the solution will look like. (In the language of [Extreme Programming \(XP\)](#), this exploratory step is called a *spike*.) Once you see the general shape of the solution, you can then use TDD to implement a more polished version.

In this section, we'll be running the tests using the **rspec** command supplied by the RSpec gem. This practice is straightforward but not ideal, and if you are a more advanced user I suggest setting up your system as described in [Section 3.6](#).

3.2.1 Test-driven development

In test-driven development, we first write a *failing* test, represented in many testing tools by the color red. We then implement code to get the test to pass, represented by the color green. Finally, if necessary, we refactor the code, changing its form (by eliminating duplication, for example) without changing its function. This cycle is known as “Red, Green, Refactor”.

We'll begin by adding some content to the Home page using test-driven development, including a top-level heading (**<h1>**) with the content **Sample App**. The first step is to generate an integration test (request spec) for our static pages:

```
$ rails generate integration_test static_pages
  invoke  rspec
  create  spec/requests/static_pages_spec.rb
```

This creates the **static_pages_spec.rb** in the **spec/requests** directory. As with most generated code, the result is not pretty, so let's open **static_pages_spec.rb** with a text editor and replace it with the contents of [Listing 3.9](#).

Listing 3.9. Code to test the contents of the Home page.

spec/requests/static_pages_spec.rb

```
require 'spec_helper'

describe "Static pages" do

  describe "Home page" do

    it "should have the content 'Sample App'" do
      visit '/static_pages/home'
      page.should have_content('Sample App')
    end
  end
end
```

The code in [Listing 3.9](#) is pure Ruby, but even if you've studied Ruby before it probably won't look very familiar. This is because RSpec uses the general malleability of Ruby to define a *domain-specific language* (DSL) built just for testing. The important point is that *you do not need to understand RSpec's syntax to be able to use RSpec*. It may seem like magic at first, but RSpec and Capybara are designed to read more or less like English, and if you follow the examples from the **generate** script and the other examples in this tutorial you'll pick it up fairly quickly.

[Listing 3.9](#) contains a **describe** block with one *example*, i.e., a block starting with **it** `"..."` **do**:

```
describe "Home page" do

  it "should have the content 'Sample App'" do
    visit '/static_pages/home'
    page.should have_content('Sample App')
  end
end
```

The first line indicates that we are describing the Home page. This is just a string, and it can be anything you want; RSpec doesn't care, but you and other human readers probably do. Then the

spec says that when you visit the Home page at `/static_pages/home`, the content should contain the words “Sample App”. As with the first line, what goes inside the quote marks is irrelevant to RSpec, and is intended to be descriptive to human readers. Then the line

```
visit '/static_pages/home'
```

uses the Capybara function `visit` to simulate visiting the URI `/static_pages/home` in a browser, while the line


```
page.should have_content('Sample App')
```

uses the `page` variable (also provided by Capybara) to test that the resulting page has the right content.

To run the test, we have several options, including some convenient but rather advanced tools discussed in [Section 3.6](#). For now, we’ll use the `rspec` command at the command line (executed with `bundle exec` to ensure that RSpec runs in the environment specified by our `Gemfile`):⁹

```
$ bundle exec rspec spec/requests/static_pages_spec.rb
```

This yields a failing test. The appearance of the result depends on your system; on my system, the red failing test appears as in [Figure 3.6](#).¹⁰ (The screenshot, which predates, the current Git branching strategy, shows work on the `master` branch instead the `static-pages` branch, but this is not cause for concern.)



```
1. ~/rails_projects/sample_app (bash)
[sample_app (master)]$ bundle exec rspec spec/requests/static_pages_spec.rb
F

Failures:

  1) StaticPages Home page should have the content 'Sample App'
     Failure/Error: page.should have_content('Sample App')
       expected there to be content "Sample App" in "SampleApp\n\nStaticPages#home\nFind me in app/views/static_pages/home.html.erb\n\n"
     # ./spec/requests/static_pages_spec.rb:9:in `block (3 levels) in <top (required)>'

Finished in 6.69 seconds
1 example, 1 failure

Failed examples:

rspec ./spec/requests/static_pages_spec.rb:7 # StaticPages Home page should have the content 'Sample App'
[sample_app (master)]$
```

Figure 3.6: A red (failing) test. [\(full size\)](#)

To get the test to pass, we'll replace the default Home page test with the HTML in [Listing 3.10](#).

Listing 3.10. Code to get a passing test for the Home page.

```
app/views/static_pages/home.html.erb
```

```
<h1>Sample App</h1>
<p>
  This is the home page for the
  <a href="http://railstutorial.org/">Ruby on Rails Tutorial</a>
  sample application.
</p>
```


This arranges for a top-level heading (`<h1>`) with the content **Sample App**, which should get the test to pass. We also include an *anchor* tag `a`, which creates links to the given URI (called an “href”, or “hypertext reference”, in the context of an anchor tag):

```
<a href="http://railstutorial.org/">Ruby on Rails Tutorial</a>
```

Now re-run the test to see the effect:

```
$ bundle exec rspec spec/requests/static_pages_spec.rb
```

On my system, the passing test appears as in [Figure 3.7](#).

A terminal window titled "1. ~/rails_projects/sample_app (ruby)" showing the execution of an RSpec test. The command entered is "bundle exec rspec spec/requests/static_pages_spec.rb". The output shows a single green dot representing a passing test, followed by "Finished in 6.62 seconds" and "1 example, 0 failures" in green text. The prompt "[sample_app (master)]\$" is visible at the end of the line.

```
1. ~/rails_projects/sample_app (ruby)
[sample_app (master)]$ bundle exec rspec spec/requests/static_pages_spec.rb
.

Finished in 6.62 seconds
1 example, 0 failures
[sample_app (master)]$
```

Figure 3.7: A green (passing) test. [\(full size\)](#)

Based on the example for the Home page, you can probably guess the analogous test and application code for the Help page. We start by testing for the relevant content, in this case the string `'Help'` ([Listing 3.11](#)).

Listing 3.11. Adding code to test the contents of the Help page.

`spec/requests/static_pages_spec.rb`

```
require 'spec_helper'

describe "Static pages" do
  describe "Home page" do
    it "should have the content 'Sample App'" do
      visit '/static_pages/home'
      page.should have_content('Sample App')
    end
  end

  describe "Help page" do
    it "should have the content 'Help'" do
      visit '/static_pages/help'
      page.should have_content('Help')
    end
  end
end
```

Then run the tests:

```
$ bundle exec rspec spec/requests/static_pages_spec.rb
```

One test should fail. (Since systems will vary, and since keeping track of how many tests there are at each stage of the tutorial is a maintenance nightmare, I'll omit the RSpec output from now on.)

The application code (which for now is raw HTML) is similar to the code in [Listing 3.10](#), as seen in [Listing 3.12](#).

Listing 3.12. Code to get a passing test for the Help page.

`app/views/static_pages/help.html.erb`

```
<h1>Help</h1>
<p>
  Get help on the Ruby on Rails Tutorial at the
  <a href="http://railstutorial.org/help">Rails Tutorial help page</a>.
  To get help on this sample app, see the
  <a href="http://railstutorial.org/book">Rails Tutorial book</a>.
</p>
```

The tests should now pass:

```
$ bundle exec rspec spec/requests/static_pages_spec.rb
```

3.2.2 Adding a page

Having seen test-driven development in action in a simple example, we'll use the same technique to accomplish the slightly more complicated task of adding a new page, namely, the About page that we intentionally left off in [Section 3.1.2](#). By writing a test and running RSpec at each step, we'll see how TDD can guide us through the development of our application code.

Red

We'll get to the Red part of the Red-Green cycle by writing a failing test for the About page. Following the models from [Listing 3.11](#), you can probably guess the right test ([Listing 3.13](#)).

Listing 3.13. Adding code to test the contents of the About page.

`spec/requests/static_pages_spec.rb`

```
require 'spec_helper'

describe "Static pages" do

  describe "Home page" do

    it "should have the content 'Sample App'" do
      visit '/static_pages/home'
      page.should have_content('Sample App')
    end
  end

  describe "Help page" do

    it "should have the content 'Help'" do
      visit '/static_pages/help'
      page.should have_content('Help')
    end
  end

  describe "About page" do

    it "should have the content 'About Us'" do
      visit '/static_pages/about'
      page.should have_content('About Us')
    end
  end
end
```

Green

Recall from [Section 3.1.2](#) that we can generate a static page in Rails by creating an action and corresponding view with the page's name. In our case, the About page will first need an action called **about** in the StaticPages controller. Having written a failing test, we can now be confident that, in getting it to pass, we will actually have created a working About page.

If you run the RSpec example using

```
$ bundle exec rspec spec/requests/static_pages_spec.rb
```

the output includes the following complaint:

```
No route matches [GET] "/static_pages/about"
```

This is a hint that we need to add `/static_pages/about` to the routes file, which we can accomplish by following the pattern in [Listing 3.5](#), as shown in [Listing 3.14](#).

Listing 3.14. Adding the `about` route.

`config/routes.rb`

```
SampleApp::Application.routes.draw do
  get "static_pages/home"
  get "static_pages/help"
  get "static_pages/about"
  .
  .
  .
end
```

Now running

```
$ bundle exec rspec spec/requests/static_pages_spec.rb
```

complains that

```
The action 'about' could not be found for StaticPagesController
```

To solve this problem, we follow the model provided by `home` and `help` from [Listing 3.6](#) by adding an `about` action in the StaticPages controller ([Listing 3.15](#)).

Listing 3.15. The StaticPages controller with added `about` action.

`app/controllers/static_pages_controller.rb`

```
class StaticPagesController < ApplicationController

  def home
  end

  def help
  end

  def about
  end
end
```

Now running

```
$ bundle exec rspec spec/requests/static_pages_spec.rb
```

says that we are missing a “template”, i.e., a view:

```
ActionView::MissingTemplate:
  Missing template static_pages/about
```

To solve this issue, we add the `about` view. This involves creating a new file called `about.html.erb` in the `app/views/static_pages` directory with the contents shown in [Listing 3.16](#).

Listing 3.16. Code for the About page.

`app/views/static_pages/about.html.erb`

```
<h1>About Us</h1>
<p>
  The <a href="http://railstutorial.org/">Ruby on Rails Tutorial</a>
  is a project to make a book and screencasts to teach web development
  with <a href="http://rubyonrails.org/">Ruby on Rails</a>. This
  is the sample application for the tutorial.
</p>
```

Running RSpec should now get us back to Green:

```
$ bundle exec rspec spec/requests/static_pages_spec.rb
```

Of course, it's never a bad idea to take a look at the page in a browser to make sure our tests aren't completely crazy ([Figure 3.8](#)).

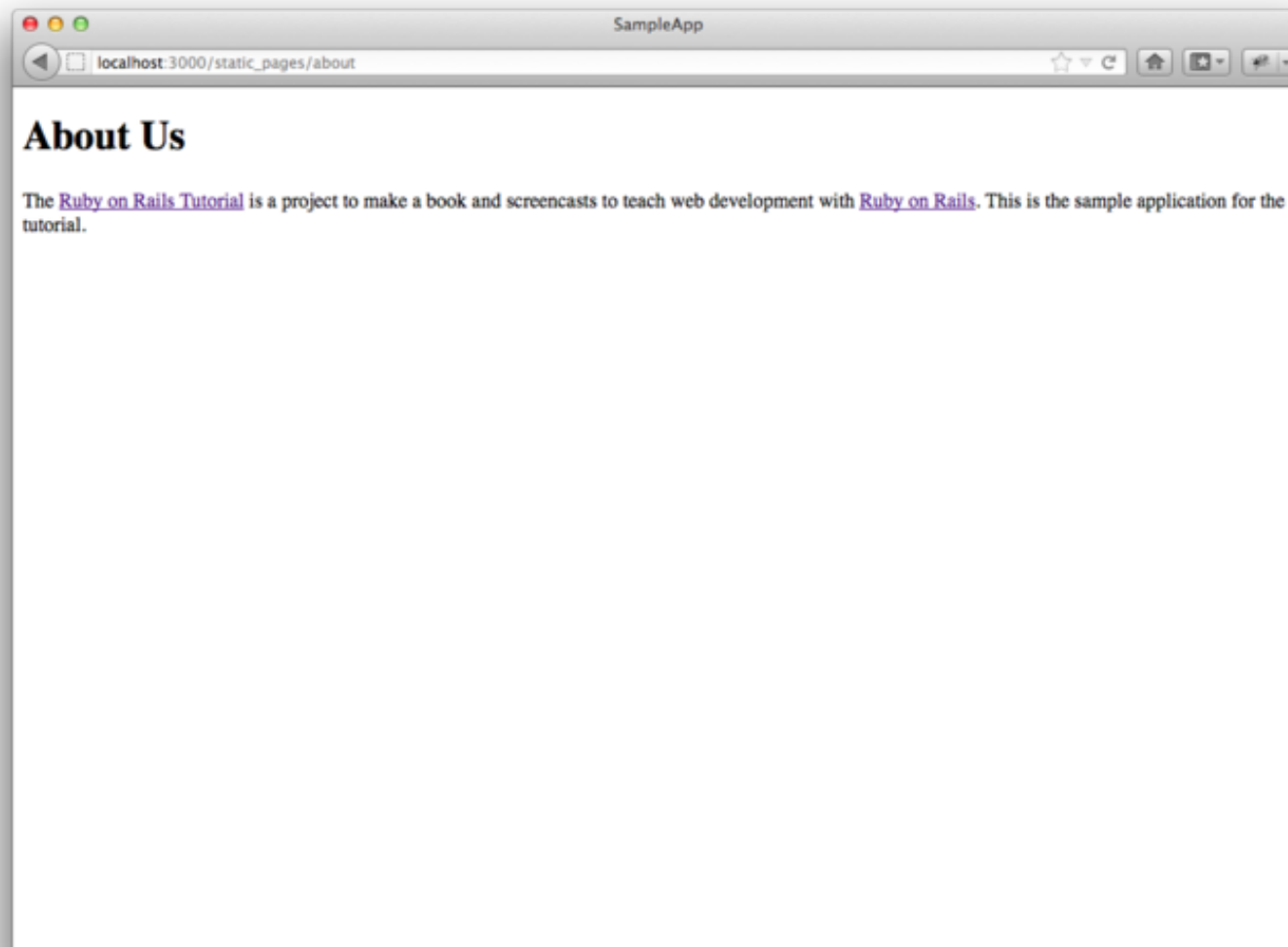


Figure 3.8: The new About page (/static_pages/about). [\(full size\)](#)

Refactor

Now that we've gotten to Green, we are free to refactor our code with confidence. Oftentimes code will start to “smell”, meaning that it gets ugly, bloated, or filled with repetition. The computer

doesn't care, of course, but humans do, so it is important to keep the code base clean by refactoring frequently. Having a good test suite is an invaluable tool in this regard, as it dramatically lowers the probability of introducing bugs while refactoring.

Our sample app is a little too small to refactor right now, but code smell seeps in at every crack, so we won't have to wait long: we'll already get busy refactoring in [Section 3.3.4](#).

3.3 Slightly dynamic pages

Now that we've created the actions and views for some static pages, we'll make them *very slightly* dynamic by adding some content that changes on a per-page basis: we'll have the title of each page change to reflect its content. Whether a changing title represents *truly* dynamic content is debatable, but in any case it lays the necessary foundation for unambiguously dynamic content in [Chapter 7](#).

If you skipped the TDD material in [Section 3.2](#), be sure to create an About page at this point using the code from [Listing 3.14](#), [Listing 3.15](#), and [Listing 3.16](#).

3.3.1 Testing a title change

Our plan is to edit the Home, Help, and About pages to make page titles that change on each page. This will involve using the `<title>` tag in our page views. Most browsers display the contents of the title tag at the top of the browser window (Google Chrome is an odd exception), and it is also important for search-engine optimization. We'll start by writing tests for the titles, then add the titles themselves, and then use a *layout* file to refactor the resulting pages and eliminate duplication.

You may have noticed that the `rails new` command already created a layout file. We'll learn its purpose shortly, but for now you should rename it before proceeding:

```
$ mv app/views/layouts/application.html.erb foobar # temporary change
```

(**mv** is a Unix command; on Windows you may need to rename the file using the file browser or the **rename** command.) You wouldn't normally do this in a real application, but it's easier to understand the purpose of the layout file if we start by disabling it.

Page	URI	Base title	Variable title
Home	/static_pages/home	"Ruby on Rails Tutorial Sample App"	"Home"
Help	/static_pages/help	"Ruby on Rails Tutorial Sample App"	"Help"
About	/static_pages/about	"Ruby on Rails Tutorial Sample App"	"About"

Table 3.1: The (mostly) static pages for the sample app.

By the end of this section, all three of our static pages will have titles of the form “Ruby on Rails Tutorial Sample App | Home”, where the last part of the title will vary depending on the page (Table 3.1). We'll build on the tests in Listing 3.13, adding title tests following the model in Listing 3.17.

Listing 3.17. A title test.

```
it "should have the right title" do
  visit '/static_pages/home'
  page.should have_selector('title',
    :text => "Ruby on Rails Tutorial Sample App | Home")
end
```


This uses the `have_selector` method, which checks for an HTML element (the “selector”) with the given content. In other words, the code

```
page.should have_selector('title',  
                          :text => "Ruby on Rails Tutorial Sample App | Home")
```

checks to see that the content inside the `title` tag is

```
"Ruby on Rails Tutorial Sample App | Home"
```

(We’ll learn in [Section 4.3.3](#) that the `:text => "..."` syntax is a *hash* using a *symbol* as the key.) It’s worth mentioning that the content need not be an exact match; any substring works as well, so that

```
page.should have_selector('title', :text => " | Home")
```

will also match the full title.

Note that in [Listing 3.17](#) we’ve broken the material inside `have_selector` into two lines; this tells you something important about Ruby syntax: Ruby doesn’t care about newlines.^{[11](#)} The *reason* I chose to break the code into pieces is that I prefer to keep lines of source code under 80 characters for legibility.^{[12](#)} As it stands, this code formatting is still rather ugly; [Section 3.5](#) has a refactoring exercise that makes them prettier, and [Section 5.3.4](#) completely rewrites the StaticPages tests to take advantage of the latest features in RSpec.

Adding new tests for each of our three static pages, following the model of [Listing 3.17](#), gives us our

new StaticPages test ([Listing 3.18](#)).

Listing 3.18. The StaticPages controller spec with title tests.

`spec/requests/static_pages_spec.rb`

```
require 'spec_helper'

describe "Static pages" do

  describe "Home page" do

    it "should have the h1 'Sample App'" do
      visit '/static_pages/home'
      page.should have_selector('h1', :text => 'Sample App')
    end

    it "should have the title 'Home'" do
      visit '/static_pages/home'
      page.should have_selector('title',
                               :text => "Ruby on Rails Tutorial Sample App | Home")
    end
  end

  describe "Help page" do

    it "should have the h1 'Help'" do
      visit '/static_pages/help'
      page.should have_selector('h1', :text => 'Help')
    end

    it "should have the title 'Help'" do
      visit '/static_pages/help'
      page.should have_selector('title',
                               :text => "Ruby on Rails Tutorial Sample App | Help")
    end
  end

  describe "About page" do

    it "should have the h1 'About Us'" do
      visit '/static_pages/about'
      page.should have_selector('h1', :text => 'About Us')
    end
  end
end
```

```
end

it "should have the title 'About Us'" do
  visit '/static_pages/about'
  page.should have_selector('title',
    :text => "Ruby on Rails Tutorial Sample App | About Us")
end
end
end
```

Note that we've changed `have_content` to the more specific `have_selector('h1', ...)`. See if you can figure out why. (*Hint*: What would happen if the title contained, say, 'Help', but the content inside `h1` tag had 'Helf' instead?)

With the tests from [Listing 3.18](#) in place, you should run

```
$ bundle exec rspec spec/requests/static_pages_spec.rb
```

to verify that our code is now Red (failing tests).

3.3.2 Passing title tests

Now we'll get our title tests to pass, and at the same time add the full HTML structure needed to make valid web pages. Let's start with the Home page ([Listing 3.19](#)), using the same basic HTML skeleton as in the "hello" page from [Listing 3.3](#).

Listing 3.19. The view for the Home page with full HTML structure.

`app/views/static_pages/home.html.erb`

```
<!DOCTYPE html>
<html>
<head>
```

```
<title>Ruby on Rails Tutorial Sample App | Home</title>
</head>
<body>
  <h1>Sample App</h1>
  <p>
    This is the home page for the
    <a href="http://railstutorial.org/">Ruby on Rails Tutorial</a>
    sample application.
  </p>
</body>
</html>
```

[Listing 3.19](#) uses the title tested for in [Listing 3.18](#):

```
<title>Ruby on Rails Tutorial Sample App | Home</title>
```

As a result, the tests for the Home page should now pass. We're still Red because of the failing Help and About tests, and we can get to Green with the code in [Listing 3.20](#) and [Listing 3.21](#).

Listing 3.20. The view for the Help page with full HTML structure.

`app/views/static_pages/help.html.erb`

```
<!DOCTYPE html>
<html>
  <head>
    <title>Ruby on Rails Tutorial Sample App | Help</title>
  </head>
  <body>
    <h1>Help</h1>
    <p>
      Get help on the Ruby on Rails Tutorial at the
      <a href="http://railstutorial.org/help">Rails Tutorial help page</a>.
      To get help on this sample app, see the
      <a href="http://railstutorial.org/book">Rails Tutorial book</a>.
    </p>
  </body>
```

```
</html>
```

Listing 3.21. The view for the About page with full HTML structure.

`app/views/static_pages/about.html.erb`

```
<!DOCTYPE html>
<html>
  <head>
    <title>Ruby on Rails Tutorial Sample App | About Us</title>
  </head>
  <body>
    <h1>About Us</h1>
    <p>
      The <a href="http://railstutorial.org/">Ruby on Rails Tutorial</a>
      is a project to make a book and screencasts to teach web development
      with <a href="http://rubyonrails.org/">Ruby on Rails</a>. This
      is the sample application for the tutorial.
    </p>
  </body>
</html>
```

3.3.3 Embedded Ruby

We’ve achieved a lot already in this section, generating three valid pages using Rails controllers and actions, but they are purely static HTML and hence don’t show off the power of Rails. Moreover, they suffer from terrible duplication:

- The page titles are almost (but not quite) exactly the same.
- “Ruby on Rails Tutorial Sample App” is common to all three titles.
- The entire HTML skeleton structure is repeated on each page.

This repeated code is a violation of the important “Don’t Repeat Yourself” (DRY) principle; in this section and the next we’ll “DRY out our code” by removing the repetition.

Paradoxically, we'll take the first step toward eliminating duplication by first adding some more: we'll make the titles of the pages, which are currently quite similar, match *exactly*. This will make it much simpler to remove all the repetition at a stroke.

The technique involves using *Embedded Ruby* in our views. Since the Home, Help, and About page titles have a variable component, we'll use a special Rails function called **provide** to set a different title on each page. We can see how this works by replacing the literal title "Home" in the `home.html.erb` view with the code in [Listing 3.22](#).

Listing 3.22. The view for the Home page with an Embedded Ruby title.

`app/views/static_pages/home.html.erb`

```
<% provide(:title, 'Home') %>
<!DOCTYPE html>
<html>
  <head>
    <title>Ruby on Rails Tutorial Sample App | <%= yield(:title) %></title>
  </head>
  <body>
    <h1>Sample App</h1>
    <p>
      This is the home page for the
      <a href="http://railstutorial.org/">Ruby on Rails Tutorial</a>
      sample application.
    </p>
  </body>
</html>
```

[Listing 3.22](#) is our first example of Embedded Ruby, also called *ERb*. (Now you know why HTML views have the file extension `.html.erb`.) ERb is the primary template system for including dynamic content in web pages.¹³ The code

```
<% provide(:title, 'Home') %>
```

indicates using `<% ... %>` that Rails should call the **provide** function and associate the string `'Home'` with the label `:title`.¹⁴ Then, in the title, we use the closely related notation `<%= ... %>` to insert the title into the template using Ruby's **yield** function:¹⁵

```
<title>Ruby on Rails Tutorial Sample App | <%= yield(:title) %></title>
```

(The distinction between the two types of embedded Ruby is that `<% ... %>` *executes* the code inside, while `<%= ... %>` executes it and *inserts* the result into the template.) The resulting page is exactly the same as before, only now the variable part of the title is generated dynamically by ERb.

We can verify that all this works by running the tests from [Section 3.3.1](#) and see that they still pass:

```
$ bundle exec rspec spec/requests/static_pages_spec.rb
```

Then we can make the corresponding replacements for the Help and About pages ([Listing 3.23](#) and [Listing 3.24](#)).

Listing 3.23. The view for the Help page with an Embedded Ruby title.

app/views/static_pages/help.html.erb

```
<% provide(:title, 'Help') %>
<!DOCTYPE html>
<html>
  <head>
    <title>Ruby on Rails Tutorial Sample App | <%= yield(:title) %></title>
  </head>
  <body>
    <h1>Help</h1>
```

```
<p>
  Get help on the Ruby on Rails Tutorial at the
  <a href="http://railstutorial.org/help">Rails Tutorial help page</a>.
  To get help on this sample app, see the
  <a href="http://railstutorial.org/book">Rails Tutorial book</a>.
</p>
</body>
</html>
```

Listing 3.24. The view for the About page with an Embedded Ruby title.

`app/views/static_pages/about.html.erb`

```
<% provide(:title, 'About Us') %>
<!DOCTYPE html>
<html>
  <head>
    <title>Ruby on Rails Tutorial Sample App | <%= yield(:title) %></title>
  </head>
  <body>
    <h1>About Us</h1>
    <p>
      The <a href="http://railstutorial.org/">Ruby on Rails Tutorial</a>
      is a project to make a book and screencasts to teach web development
      with <a href="http://rubyonrails.org/">Ruby on Rails</a>. This
      is the sample application for the tutorial.
    </p>
  </body>
</html>
```

3.3.4 Eliminating duplication with layouts

Now that we've replaced the variable part of the page titles with ERb, each of our pages looks something like this:

```
<% provide(:title, 'Foo') %>
<!DOCTYPE html>
```



```
<html>
  <head>
    <title>Ruby on Rails Tutorial Sample App | <%= yield(:title) %></title>
  </head>
  <body>
    Contents
  </body>
</html>
```

In other words, *all* our pages are identical in structure, including the contents of the title tag, with the sole exception of the material inside the **body** tag.

In order to factor out this common structure, Rails comes with a special *layout* file called **application.html.erb**, which we renamed in [Section 3.3.1](#) and which we'll now restore:

```
$ mv foobar app/views/layouts/application.html.erb
```

To get the layout to work, we have to replace the default title with the Embedded Ruby from the examples above:

```
<title>Ruby on Rails Tutorial Sample App | <%= yield(:title) %></title>
```

The resulting layout appears in [Listing 3.25](#).

Listing 3.25. The sample application site layout.

app/views/layouts/application.html.erb

```
<!DOCTYPE html>
<html>
  <head>
    <title>Ruby on Rails Tutorial Sample App | <%= yield(:title) %></title>
```

```
<%= stylesheet_link_tag    "application", :media => "all" %>
<%= javascript_include_tag "application" %>
<%= csrf_meta_tags %>
</head>
<body>
  <%= yield %>
</body>
</html>
```

Note here the special line

```
<%= yield %>
```

This code is responsible for inserting the contents of each page into the layout. It's not important to know exactly how this works; what matters is that using this layout ensures that, for example, visiting the page `/static_pages/home` converts the contents of `home.html.erb` to HTML and then inserts it in place of `<%= yield %>`.

It's also worth noting that the default Rails layout includes several additional lines:

```
<%= stylesheet_link_tag    "application", :media => "all" %>
<%= javascript_include_tag "application" %>
<%= csrf_meta_tags %>
```

This code arranges to include the application stylesheet and JavaScript, which are part of the asset pipeline ([Section 5.2.1](#)), together with the Rails method `csrf_meta_tags`, which prevents [cross-site request forgery](#) (CSRF), a type of malicious web attack.

Of course, the views in [Listing 3.22](#), [Listing 3.23](#), and [Listing 3.24](#) are still filled with all the HTML structure included in the layout, so we have to remove it, leaving only the interior contents. The

resulting cleaned-up views appear in [Listing 3.26](#), [Listing 3.27](#), and [Listing 3.28](#).

Listing 3.26. The Home page with HTML structure removed.

`app/views/static_pages/home.html.erb`

```
<% provide(:title, 'Home') %>
<h1>Sample App</h1>
<p>
  This is the home page for the
  <a href="http://railstutorial.org/">Ruby on Rails Tutorial</a>
  sample application.
</p>
```

Listing 3.27. The Help page with HTML structure removed.

`app/views/static_pages/help.html.erb`

```
<% provide(:title, 'Help') %>
<h1>Help</h1>
<p>
  Get help on the Ruby on Rails Tutorial at the
  <a href="http://railstutorial.org/help">Rails Tutorial help page</a>.
  To get help on this sample app, see the
  <a href="http://railstutorial.org/book">Rails Tutorial book</a>.
</p>
```

Listing 3.28. The About page with HTML structure removed.

`app/views/static_pages/about.html.erb`

```
<% provide(:title, 'About Us') %>
<h1>About Us</h1>
<p>
  The <a href="http://railstutorial.org/">Ruby on Rails Tutorial</a>
  is a project to make a book and screencasts to teach web development
  with <a href="http://rubyonrails.org/">Ruby on Rails</a>. This
  is the sample application for the tutorial.
</p>
```

With these views defined, the Home, Help, and About pages are the same as before, but they have much less duplication. Verifying that the test suite still passes gives us confidence that this code refactoring was successful:

```
$ bundle exec rspec spec/requests/static_pages_spec.rb
```

3.4 Conclusion

Seen from the outside, this chapter hardly accomplished anything: we started with static pages, and ended with... *mostly* static pages. But appearances are deceiving: by developing in terms of Rails controllers, actions, and views, we are now in a position to add arbitrary amounts of dynamic content to our site. Seeing exactly how this plays out is the task for the rest of this tutorial.

Before moving on, let's take a minute to commit our changes and merge them into the master branch. Back in [Section 3.1.2](#) we created a Git branch for the development of static pages. If you haven't been making commits as we've been moving along, first make a commit indicating that we've reached a stopping point:

```
$ git add .  
$ git commit -m "Finish static pages"
```

Then merge the changes back into the master branch using the same technique as in [Section 1.3.5](#):

```
$ git checkout master  
$ git merge static-pages
```

Once you reach a stopping point like this, it's usually a good idea to push your code up to a remote repository (which, if you followed the steps in [Section 1.3.4](#), will be GitHub):

```
$ git push
```

If you like, at this point you can even deploy the updated application to Heroku:

```
$ git push heroku
```

3.5 Exercises

1. Make a Contact page for the sample app. First write a test for the existence of a page at the URI `/static_pages/contact`. (*Hint*: Test for the right title.) Then write a second test for the title “Ruby on Rails Tutorial Sample App | Contact”. Get your tests to pass, and then fill in the Contact page with the content from [Listing 3.29](#). (This exercise is solved as part of [Section 5.3](#).)
2. You may have noticed some repetition in the StaticPages controller spec ([Listing 3.18](#)). In particular, the base title, “Ruby on Rails Tutorial Sample App”, is the same for every title test. Using the RSpec `let` function, which creates a variable corresponding to its argument, verify that the tests in [Listing 3.30](#) still pass. [Listing 3.30](#) introduces *string interpolation*, which is covered further in [Section 4.2.2](#).
3. **(advanced)** As noted on the [Heroku page on using sqlite3 for development](#), it's a good idea to use the same database in development, test, and production environments to minimize the possibility of subtle incompatibilities. Follow the [Heroku instructions for local PostgreSQL installation](#) to install the PostgreSQL database on your local system. Update your `Gemfile` to eliminate the `sqlite3` gem and use the `pg` gem exclusively, as shown in

[Listing 3.31](#). You will also have to learn about the `config/database.yml` file and how to run PostgreSQL locally. Your goal should be to create and configure both the development database and the test database to use PostgreSQL. **Warning:** You may find this exercise challenging, and I recommend it only for advanced users. If you get stuck, don't hesitate to skip it; as noted previously, the sample application developed in this tutorial is fully compatible with both SQLite and PostgreSQL.

Listing 3.29. Code for a proposed Contact page.

`app/views/static_pages/contact.html.erb`

```
<% provide(:title, 'Contact') %>
<h1>Contact</h1>
<p>
  Contact Ruby on Rails Tutorial about the sample app at the
  <a href="http://railstutorial.org/contact">contact page</a>.
</p>
```

Listing 3.30. The StaticPages controller spec with a base title.

`spec/requests/static_pages_spec.rb`

```
require 'spec_helper'

describe "Static pages" do

  let(:base_title) { "Ruby on Rails Tutorial Sample App" }

  describe "Home page" do

    it "should have the h1 'Sample App'" do
      visit '/static_pages/home'
      page.should have_selector('h1', :text => 'Sample App')
    end

    it "should have the title 'Home'" do
      visit '/static_pages/home'
      page.should have_selector('title', :text => "#{base_title} | Home")
    end
  end
end
```

```

end

describe "Help page" do

  it "should have the h1 'Help'" do
    visit '/static_pages/help'
    page.should have_selector('h1', :text => 'Help')
  end

  it "should have the title 'Help'" do
    visit '/static_pages/help'
    page.should have_selector('title', :text => "#{base_title} | Help")
  end
end

describe "About page" do

  it "should have the h1 'About Us'" do
    visit '/static_pages/about'
    page.should have_selector('h1', :text => 'About Us')
  end

  it "should have the title 'About Us'" do
    visit '/static_pages/about'
    page.should have_selector('title', :text => "#{base_title} | About Us")
  end
end

describe "Contact page" do

  it "should have the h1 'Contact'" do
    visit '/static_pages/contact'
    page.should have_selector('h1', :text => 'Contact')
  end

  it "should have the title 'Contact'" do
    visit '/static_pages/contact'
    page.should have_selector('title', :text => "#{base_title} | Contact")
  end
end
end

```

Listing 3.31. The **Gemfile** needed to use PostgreSQL instead of SQLite.

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

gem 'rails', '3.2.8'
gem 'pg', '0.12.2'

group :development, :test do
  gem 'rspec-rails', '2.11.0'
end

# Gems used only for assets and not required
# in production environments by default.
group :assets do
  gem 'sass-rails', '3.2.5'
  gem 'coffee-rails', '3.2.2'
  gem 'uglifier', '1.2.3'
end

gem 'jquery-rails', '2.0.2'

group :test do
  gem 'capybara', '1.1.2'
end
```

3.6 Advanced setup

As mentioned briefly in [Section 3.2](#), using the **rspec** command directly is not ideal. In this section, we'll first discuss a method to eliminate the necessity of typing **bundle exec**, and then set up testing setup to automate the running of the test suite using Guard ([Section 3.6.2](#)) and, optionally, Spork ([Section 3.6.3](#)). Finally, we'll mention a method for running tests directly inside Sublime Text, a technique especially useful when used in concert with Spork.

This section should only be attempted by fairly advanced users and can be skipped without loss of continuity. Among other things, this material is likely to go out of date faster than the rest of the tutorial, so you shouldn't expect everything on your system to match the examples exactly, and you may have to Google around to get everything to work.

3.6.1 Eliminating `bundle exec`

As mentioned briefly in [Section 3.2.1](#), it is necessary in general to prefix commands such as `rake` or `rspec` with `bundle exec` so that the programs run in the exact gem environment specified by the `Gemfile`. (For technical reasons, the only exception to this is the `rails` command itself.) This practice is rather cumbersome, and in this section we discuss two ways to eliminate its necessity.

RVM Bundler integration

The first and preferred method is to use RVM, which includes Bundler integration as of version 1.11. You can verify that you have a sufficiently up-to-date version of RVM as follows:

```
$ rvm get head && rvm reload
$ rvm -v

rvm 1.15.6 (master)
```

As long as the version number is 1.11.x or greater, installed gems will automatically be executed in the proper Bundler environment, so that you can write (for example)

```
$ rspec spec/
```

and omit the leading `bundle exec`. If this is the case, you should skip the rest of this section.

If for any reason you are restricted to an earlier version of RVM, you can still eliminate `bundle exec` by using [RVM Bundler integration](#)¹⁶ to configure the Ruby Version Manager to include the proper executables automatically in the local environment. The steps are simple if somewhat mysterious. First, run these two commands:

```
$ rvm get head && rvm reload
$ chmod +x $rvm_path/hooks/after_cd_bundler
```

Then run these:

```
$ cd ~/rails_projects/sample_app
$ bundle install --without production --binstubs=./bundler_stubs
```

Together, these commands combine RVM and Bundler magic to ensure that commands such as **rake** and **rspec** are automatically executed in the right environment. Since these files are specific to your local setup, you should add the **bundler_stubs** directory to your **.gitignore** file ([Listing 3.32](#)).

Listing 3.32. Adding **bundler_stubs** to the **.gitignore** file.

```
# Ignore bundler config
/.bundle

# Ignore the default SQLite database.
/db/*.sqlite3

# Ignore all logfiles and tempfiles.
/log/*.log
/tmp

# Ignore other unneeded files.
doc/
*.swp
*~
.project
.DS_Store
bundler_stubs/
```

If you add another executable (such as **guard** in [Section 3.6.2](#)), you should re-run the **bundle install** command:

```
$ bundle install --binstubs=./bundler_stubs
```

binstubs

If you're not using RVM, you can still avoid typing **bundle exec**. Bundler allows the creation of the associated binaries as follows:

```
$ bundle --binstubs
```

(In fact, this step, with a different target directory, is also used when using RVM.) This command creates all the necessary executables in the **bin/** directory of the application, so that we can now run the test suite as follows:

```
$ bin/rspec spec/
```

The same goes for **rake**, etc.:

```
$ bin/rake db:migrate
```

If you add another executable (such as **guard** in [Section 3.6.2](#)), you should re-run the **bundle --binstubs** command.

For the sake of readers who skip this section, the rest of this tutorial will err on the side of caution and explicitly use `bundle exec`, but of course you should feel free to use the more compact version if your system is properly configured.

3.6.2 Automated tests with Guard

One annoyance associated with using the `rspec` command is having to switch to the command line and run the tests by hand. (A second annoyance, the slow start-up time of the test suite, is addressed in [Section 3.6.3](#).) In this section, we'll show how to use [Guard](#) to automate the running of the tests. Guard monitors changes in the filesystem so that, for example, when we change the `static_pages_spec.rb` file only those tests get run. Even better, we can configure Guard so that when, say, the `home.html.erb` file is modified, the `static_pages_spec.rb` automatically runs.

First we add `guard-rspec` to the `Gemfile` ([Listing 3.33](#)).

Listing 3.33. A `Gemfile` for the sample app, including Guard.

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

gem 'rails', '3.2.8'

group :development, :test do
  gem 'sqlite3', '1.3.5'
  gem 'rspec-rails', '2.11.0'
  gem 'guard-rspec', '0.5.5'
end

# Gems used only for assets and not required
# in production environments by default.
group :assets do
  gem 'sass-rails', '3.2.5'
  gem 'coffee-rails', '3.2.2'
  gem 'uglifier', '1.2.3'
end
```

```
gem 'jquery-rails', '2.0.2'

group :test do
  gem 'capybara', '1.1.2'
  # System-dependent gems
end

group :production do
  gem 'pg', '0.12.2'
end
```

Then we have to replace the comment at the end of the test group with some system-dependent gems (OS X users may have to install [Growl](#) and [growlnotify](#) as well):

```
# Test gems on Macintosh OS X
group :test do
  gem 'capybara', '1.1.2'
  gem 'rb-fsevent', '0.9.1', :require => false
  gem 'growl', '1.0.3'
end
```

```
# Test gems on Linux
group :test do
  gem 'capybara', '1.1.2'
  gem 'rb-inotify', '0.8.8'
  gem 'libnotify', '0.5.9'
end
```

```
# Test gems on Windows
group :test do
  gem 'capybara', '1.1.2'
  gem 'rb-fchange', '0.0.5'
  gem 'rb-notifu', '0.0.4'
  gem 'win32console', '1.3.0'
```

```
end
```

We next install the gems by running **bundle install**:

```
$ bundle install
```

Then initialize Guard so that it works with RSpec:

```
$ bundle exec guard init rspec
Writing new Guardfile to /Users/mhartl/rails_projects/sample_app/Guardfile
rspec guard added to Guardfile, feel free to edit it
```

Now edit the resulting **Guardfile** so that Guard will run the right tests when the integration tests and views are updated ([Listing 3.34](#)).

Listing 3.34. Additions to the default **Guardfile**.

```
require 'active_support/core_ext'

guard 'rspec', :version => 2, :all_after_pass => false do
  .
  .
  .
  watch(%r{^app/controllers/(.+)_controller\.rb$}) do |m|
    ["spec/routing/#{m[1]}_routing_spec.rb",
     "spec/#{m[2]}s/#{m[1]}_#{m[2]}_spec.rb",
     "spec/acceptance/#{m[1]}_spec.rb",
     (m[1][/_pages/] ? "spec/requests/#{m[1]}_spec.rb" :
      "spec/requests/#{m[1].singularize}_pages_spec.rb")]
  end
  watch(%r{^app/views/(.+)/}) do |m|
    (m[1][/_pages/] ? "spec/requests/#{m[1]}_spec.rb" :
     "spec/requests/#{m[1].singularize}_pages_spec.rb")
  end
end
```

```
end  
.  
.  
.  
end
```

Here the line

```
guard 'rspec', :version => 2, :all_after_pass => false do
```

ensures that Guard doesn't run all the tests after a failing test passes (to speed up the Red-Green-Refactor cycle).

We can now start **guard** as follows:

```
$ bundle exec guard
```

To eliminate the need to prefix the command with **bundle exec**, re-follow the steps in [Section 3.6.1](#).

By the way, if you get a Guard error complaining about the absence of a **spec/routing** directory, you can fix it by creating an empty one:

```
$ mkdir spec/routing
```

3.6.3 Speeding up tests with Spork

When running `bundle exec rspec`, you may have noticed that it takes several seconds just to start running the tests, but once they start running they finish quickly. This is because each time RSpec runs the tests it has to reload the entire Rails environment. The [Spork test server¹⁷](#) aims to solve this problem. Spork loads the environment *once*, and then maintains a pool of processes for running future tests. Spork is particularly useful when combined with Guard ([Section 3.6.2](#)).

The first step is to add the `spork` gem dependency to the `Gemfile` ([Listing 3.35](#)).

Listing 3.35. A `Gemfile` for the sample app.

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

gem 'rails', '3.2.8'
.
.
.
group :test do
  .
  .
  .
  gem 'guard-spork', '0.3.2'
  gem 'spork', '0.9.0'
end
```

Then install Spork using `bundle install`:

```
$ bundle install
```

Next, bootstrap the Spork configuration:

```
$ bundle exec spork --bootstrap
```


Now we need to edit the RSpec configuration file, located in `spec/spec_helper.rb`, so that the environment gets loaded in a `prefork` block, which arranges for it to be loaded only once ([Listing 3.36](#)).

Listing 3.36. Adding environment loading to the `Spork.prefork` block.

`spec/spec_helper.rb`

```
require 'rubygems'
require 'spork'

Spork.prefork do
  # Loading more in this block will cause your tests to run faster. However,
  # if you change any configuration or code from libraries loaded here, you'll
  # need to restart spork for it take effect.
  # This file is copied to spec/ when you run 'rails generate rspec:install'
  ENV["RAILS_ENV"] ||= 'test'
  require File.expand_path("../../config/environment", __FILE__)
  require 'rspec/rails'
  require 'rspec/autorun'

  # Requires supporting ruby files with custom matchers and macros, etc,
  # in spec/support/ and its subdirectories.
  Dir[Rails.root.join("spec/support/**/*.rb")].each {|f| require f}

  RSpec.configure do |config|
    # == Mock Framework
    #
    # If you prefer to use mocha, flexmock or RR, uncomment the appropriate line:
    #
    # config.mock_with :mocha
    # config.mock_with :flexmock
    # config.mock_with :rr
    config.mock_with :rspec

    # Remove this line if you're not using ActiveRecord or ActiveRecord fixtures
    config.fixture_path = "#{::Rails.root}/spec/fixtures"

    # If you're not using ActiveRecord, or you'd prefer not to run each of your
    # examples within a transaction, remove the following line or assign false
    # instead of true.
```

```
config.use_transactional_fixtures = true

# If true, the base class of anonymous controllers will be inferred
# automatically. This will be the default behavior in future versions of
# rspec-rails.
config.infer_base_class_for_anonymous_controllers = false
end
end

Spork.each_run do
  # This code will be run each time you run your specs.
end
```

Before running Spork, we can get a baseline for the testing overhead by timing our test suite as follows:

```
$ time bundle exec rspec spec/requests/static_pages_spec.rb
.....

6 examples, 0 failures

real    0m8.633s
user    0m7.240s
sys     0m1.068s
```

Here the test suite takes more than seven seconds to run even though the actual tests run in under a tenth of a second. To speed this up, we can open a dedicated terminal window, navigate to the application root directory, and then start a Spork server:

```
$ bundle exec spork
Using RSpec
Loading Spork.prefork block...
Spork is ready and listening on 8989!
```

(To eliminate the need to prefix the command with **bundle exec**, re-follow the steps in [Section 3.6.1](#).) In another terminal window, we can now run our test suite with the `--drb` (“distributed Ruby”) option and verify that the environment-loading overhead is greatly reduced:

```
$ time bundle exec rspec spec/requests/static_pages_spec.rb --drb
.....

6 examples, 0 failures

real    0m2.649s
user    0m1.259s
sys     0m0.258s
```

It’s inconvenient to have to include the `--drb` option every time we run **rspec**, so I recommend adding it to the `.rspec` file in the application’s root directory, as shown in [Listing 3.37](#).

Listing 3.37. Configuring RSpec to automatically use Spork.

.rspec

```
--colour
--drb
```

One word of advice when using Spork: after changing a file included in the prefork loading (such as **routes.rb**), you will have to restart the Spork server to load the new Rails environment. If your tests are failing when you think they should be passing, quit the Spork server with `Control-C` and restart it:

```
$ bundle exec spork
Using RSpec
Loading Spork.prefork block...
Spork is ready and listening on 8989!
```

```
^C
$ bundle exec spork
```

Guard with Spork

Spork is especially useful when used with Guard, which we can arrange as follows:

```
$ bundle exec guard init spork
```

We then need to change the **Guardfile** as in [Listing 3.38](#).

Listing 3.38. The **Guardfile** updated for Spork.

```
require 'active_support/core_ext'

guard 'spork', :rspec_env => { 'RAILS_ENV' => 'test' } do
  watch('config/application.rb')
  watch('config/environment.rb')
  watch(%r{^config/environments/.+\.rb$})
  watch(%r{^config/initializers/.+\.rb$})
  watch('Gemfile')
  watch('Gemfile.lock')
  watch('spec/spec_helper.rb')
  watch('test/test_helper.rb')
  watch('spec/support/')
end

guard 'rspec', :version => 2, :all_after_pass => false, :cli => '--drb' do
  .
  .
  .
end
```

Note that we've updated the arguments to **guard** to include `:cli => --drb`, which ensures that

Guard uses the command-line interface (cli) to the Spork server. We've also added a command to watch the `spec/support/` directory, which we'll start modifying in [Chapter 5](#).

With that configuration in place, we can start Guard and Spork at the same time with the `guard` command:

```
$ bundle exec guard
```

Guard automatically starts a Spork server, dramatically reducing the overhead each time a test gets run.

A well-configured testing environment with Guard, Spork, and (optionally) test notifications makes test-driven development positively addictive. See the [Rails Tutorial screencasts](#)¹⁸ for more information.

3.6.4 Tests inside Sublime Text

If you're using Sublime Text, there is a powerful set of helper commands to run tests directly inside the editor. To get them working, follow the instructions for your platform at [Sublime Text 2 Ruby Tests](#).¹⁹ On my platform (Macintosh OS X), I can install the commands as follows:

```
$ cd ~/Library/Application\ Support/Sublime\ Text\ 2/Packages
$ git clone https://github.com/maltize/sublime-text-2-ruby-tests.git RubyTest
```

You may also want to follow the setup instructions for [Rails Tutorial Sublime Text](#) at this time.²⁰

After restarting Sublime Text, the RubyTest package supplies the following commands:

- **Command-Shift-R:** run a single test (if run on an `it` block) or group of tests (if run on a `describe` block)
- **Command-Shift-E:** run the last test(s)
- **Command-Shift-T:** run all the tests in current file

Because test suites can become quite slow even for relatively small projects, being able to run one test (or a small group of tests) at a time can be a huge win. Even a single test requires the same Rails environment overhead, of course, which is why these commands are perfectly complemented by Spork: running a single test eliminates the overhead of running the entire test file, while running Spork eliminates the overhead of starting the test environment. Here is the sequence I recommend:

1. Start Spork in a terminal window.
2. Write a single test or small group of tests.
3. Run Command-Shift-R to verify that the test or test group is red.
4. Write the corresponding application code.
5. Run Command-Shift-E to run the same test/group again, verifying that it's green.
6. Repeat steps 2–5 as necessary.
7. When reaching a natural stopping point (such as before a commit), run `rspec spec/` at the command line to confirm that the entire test suite is still green.

Even with the ability to run tests inside of Sublime Text, I still sometimes prefer using Guard, but at this point my bread-and-butter TDD technique is the one enumerated above.

[« Chapter 2 A demo app](#)

[Chapter 4 Rails-flavored Ruby »](#)

1. The successor to *Webrat*, Capybara is named after the world's [largest rodent](#). ↑
2. In fact, you can even leave off `install`. The `bundle` command by itself is an alias for `bundle install`. ↑
3. As before, you may find the augmented file from [Listing 1.7](#) to be more convenient depending on your system. ↑
4. https://github.com/railstutorial/sample_app_2nd_ed ↑
5. In fact, Rails ensures that requests for such files never hit the main Rails stack; they are delivered directly from the filesystem. (See [The Rails 3 Way](#) for more details.) ↑
6. As usual, replace `subl` with the command for your text editor. ↑
7. HTML changes with time; by explicitly making a doctype declaration we make it likelier that browsers will render our pages properly in the future. The extremely simple doctype `<!DOCTYPE html>` is characteristic of the latest HTML standard, HTML5. ↑
8. Our method for making static pages is probably the simplest, but it's not the only way. The optimal method really depends on your needs; if you expect a *large* number of static pages, using a StaticPages controller can get quite cumbersome, but in our sample app we'll only need a few. See [this blog post on simple pages at has many :through](#) for a survey of techniques for making static pages with Rails. *Warning:* The discussion is fairly advanced, so you might want to wait a while before trying to understand it. ↑
9. Running `bundle exec` every time is rather cumbersome; see [Section 3.6](#) for some options to eliminate it. ↑
10. I actually use a dark background for both my terminal and editor, but the light background looks better in the screenshots. ↑
11. A newline is what comes at the end of a line, thereby starting a new line. In code, it is represented

by the character `\n`. ↑

12. Actually *counting* columns could drive you crazy, which is why many text editors have a visual aid to help you. For example, if you take a look back at [Figure 1.1](#), you'll see a small vertical line on the right to help keep code under 80 characters. (It's actually at 78 columns, which gives you a little margin for error.) If you use TextMate, you can find this feature under View > Wrap Column > 78. In Sublime Text, you can use View > Ruler > 78 or View > Ruler > 80. ↑
13. There is a second popular template system called [Haml](#), which I personally love, but it's not *quite* standard enough yet for use in an introductory tutorial. ↑
14. Experienced Rails developers might have expected the use of `content_for` at this point, but it doesn't work well with the asset pipeline. The `provide` function is its replacement. ↑
15. If you've studied Ruby before, you might suspect that Rails is *yielding* the contents to a block, and your suspicion would be correct. But you don't need to know this to develop applications with Rails. ↑
16. <http://rvm.io/integration/bundler/> ↑
17. A *spork* is a combination spoon-fork. The project's name is a pun on Spork's use of [POSIX forks](#). ↑
18. <http://railstutorial.org/screencasts> ↑
19. <https://github.com/maltize/sublime-text-2-ruby-tests> ↑
20. https://github.com/mhartl/rails_tutorial_sublime_text ↑