



# [Ruby on Rails Tutorial](#)

[by Michael Hartl](#)



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



**Buy Print Edition**

**Buy Screencasts**



Rails 3 • [Rails 2.3](#)

[Rails Tutorial Affiliate Program—50% commissions](#)

[Bonus Screencast](#)

[on Haml](#)

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

## Ruby on Rails Tutorial

## Learn Rails by Example

Michael Hartl

## Contents

1. [Chapter 1 From zero to deploy](#)
2.
  1. [1.1 Introduction](#)
  2.
    1. [1.1.1 Comments for various readers](#)
    2. [1.1.2 “Scaling” Rails](#)
    3. [1.1.3 Conventions in this book](#)
  3. [1.2 Up and running](#)
  4.
    1. [1.2.1 Development environments](#)
    2.
      1. [IDEs](#)
      2. [Text editors and command lines](#)
      3. [Browsers](#)
      4. [A note about tools](#)

3. [1.2.2 Ruby, RubyGems, Rails, and Git](#)
4.
  1. [Rails Installer \(Windows\)](#)
  2. [Install Git](#)
  3. [Install Ruby](#)
  4. [Install RubyGems](#)
  5. [Install Rails](#)
5. [1.2.3 The first application](#)
6. [1.2.4 Bundler](#)
7. [1.2.5 rails server](#)
8. [1.2.6 Model-view-controller \(MVC\)](#)
5. [1.3 Version control with Git](#)
6.
  1. [1.3.1 Installation and setup](#)
  2.
    1. [First-time system setup](#)
    2. [First-time repository setup](#)
  3. [1.3.2 Adding and committing](#)
  4. [1.3.3 What good does Git do you?](#)
  5. [1.3.4 GitHub](#)
  6. [1.3.5 Branch, edit, commit, merge](#)
  7.
    1. [Branch](#)
    2. [Edit](#)
    3. [Commit](#)
    4. [Merge](#)
    5. [Push](#)
7. [1.4 Deploying](#)
8.
  1. [1.4.1 Heroku setup](#)
  2. [1.4.2 Heroku deployment, step one](#)
  3. [1.4.3 Heroku deployment, step two](#)
  4. [1.4.4 Heroku commands](#)
9. [1.5 Conclusion](#)
3. [Chapter 2 A demo app](#)
4.
  1. [2.1 Planning the application](#)
  2.
    1. [2.1.1 Modeling users](#)
    2. [2.1.2 Modeling microposts](#)
  3. [2.2 The Users resource](#)
  4.
    1. [2.2.1 A user tour](#)
    2. [2.2.2 MVC in action](#)
    3. [2.2.3 Weaknesses of this Users resource](#)
  5. [2.3 The Microposts resource](#)
  6.
    1. [2.3.1 A micropost microtour](#)
    2. [2.3.2 Putting the \*micro\* in microposts](#)
    3. [2.3.3 A user has many microposts](#)
    4. [2.3.4 Inheritance hierarchies](#)
    5. [2.3.5 Deploying the demo app](#)
  7. [2.4 Conclusion](#)
  5. [Chapter 3 Mostly static pages](#)
  6.
    1. [3.1 Static pages](#)
    2.
      1. [3.1.1 Truly static pages](#)
      2. [3.1.2 Static pages with Rails](#)
    3. [3.2 Our first tests](#)
    4.
      1. [3.2.1 Testing tools](#)
      2.
        1. [Autotest](#)

3. [3.2.2 TDD: Red, Green, Refactor](#)
    4.
      1. [Spork](#)
      2. [Red](#)
      3. [Green](#)
      4. [Refactor](#)
  5. [3.3 Slightly dynamic pages](#)
  6.
    1. [3.3.1 Testing a title change](#)
    2. [3.3.2 Passing title tests](#)
    3. [3.3.3 Instance variables and Embedded Ruby](#)
    4. [3.3.4 Eliminating duplication with layouts](#)
  7. [3.4 Conclusion](#)
  8. [3.5 Exercises](#)
7. [Chapter 4 Rails-flavored Ruby](#)
8.
  1. [4.1 Motivation](#)
  2.
    1. [4.1.1 A title helper](#)
    2. [4.1.2 Cascading Style Sheets](#)
  3. [4.2 Strings and methods](#)
  4.
    1. [4.2.1 Comments](#)
    2. [4.2.2 Strings](#)
    3.
      1. [Printing](#)
      2. [Single-quoted strings](#)
    4. [4.2.3 Objects and message passing](#)
    5. [4.2.4 Method definitions](#)
    6. [4.2.5 Back to the title helper](#)
  5. [4.3 Other data structures](#)
  6.
    1. [4.3.1 Arrays and ranges](#)
    2. [4.3.2 Blocks](#)
    3. [4.3.3 Hashes and symbols](#)
    4. [4.3.4 CSS revisited](#)
  7. [4.4 Ruby classes](#)
  8.
    1. [4.4.1 Constructors](#)
    2. [4.4.2 Class inheritance](#)
    3. [4.4.3 Modifying built-in classes](#)
    4. [4.4.4 A controller class](#)
    5. [4.4.5 A user class](#)
  9. [4.5 Exercises](#)
9. [Chapter 5 Filling in the layout](#)
10.
  1. [5.1 Adding some structure](#)
  2.
    1. [5.1.1 Site navigation](#)
    2. [5.1.2 Custom CSS](#)
    3. [5.1.3 Partials](#)
  3. [5.2 Layout links](#)
  4.
    1. [5.2.1 Integration tests](#)
    2. [5.2.2 Rails routes](#)
    3. [5.2.3 Named routes](#)
  5. [5.3 User signup: A first step](#)
  6.
    1. [5.3.1 Users controller](#)
    2. [5.3.2 Signup URL](#)
  7. [5.4 Conclusion](#)
  8. [5.5 Exercises](#)
11. [Chapter 6 Modeling and viewing users, part I](#)

12.
  1. [6.1 User model](#)
  2.
    1. [6.1.1 Database migrations](#)
    2. [6.1.2 The model file](#)
    3.
      1. [Model annotation](#)
      2. [Accessible attributes](#)
    4. [6.1.3 Creating user objects](#)
    5. [6.1.4 Finding user objects](#)
    6. [6.1.5 Updating user objects](#)
  3. [6.2 User validations](#)
  4.
    1. [6.2.1 Validating presence](#)
    2. [6.2.2 Length validation](#)
    3. [6.2.3 Format validation](#)
    4. [6.2.4 Uniqueness validation](#)
    5.
      1. [The uniqueness caveat](#)
  5. [6.3 Viewing users](#)
  6.
    1. [6.3.1 Debug and Rails environments](#)
    2. [6.3.2 User model, view, controller](#)
    3. [6.3.3 A Users resource](#)
    4.
      1. [params in debug](#)
  7. [6.4 Conclusion](#)
  8. [6.5 Exercises](#)
13. [Chapter 7 Modeling and viewing users, part II](#)
14.
  1. [7.1 Insecure passwords](#)
  2.
    1. [7.1.1 Password validations](#)
    2. [7.1.2 A password migration](#)
    3. [7.1.3 An Active Record callback](#)
  3. [7.2 Secure passwords](#)
  4.
    1. [7.2.1 A secure password test](#)
    2. [7.2.2 Some secure password theory](#)
    3. [7.2.3 Implementing has\\_password?](#)
    4. [7.2.4 An authenticate method](#)
  5. [7.3 Better user views](#)
  6.
    1. [7.3.1 Testing the user show page \(with factories\)](#)
    2. [7.3.2 A name and a Gravatar](#)
    3.
      1. [A Gravatar helper](#)
    4. [7.3.3 A user sidebar](#)
  7. [7.4 Conclusion](#)
  8.
    1. [7.4.1 Git commit](#)
    2. [7.4.2 Heroku deploy](#)
  9. [7.5 Exercises](#)
15. [Chapter 8 Sign up](#)
16.
  1. [8.1 Signup form](#)
  2.
    1. [8.1.1 Using form\\_for](#)
    2. [8.1.2 The form HTML](#)
  3. [8.2 Signup failure](#)
  4.
    1. [8.2.1 Testing failure](#)
    2. [8.2.2 A working form](#)
    3. [8.2.3 Signup error messages](#)
    4. [8.2.4 Filtering parameter logging](#)
  5. [8.3 Signup success](#)
  6.
    1. [8.3.1 Testing success](#)

2. [8.3.2 The finished signup form](#)
    3. [8.3.3 The flash](#)
    4. [8.3.4 The first signup](#)
  7. [8.4 RSpec integration tests](#)
  8.
    1. [8.4.1 Integration tests with style](#)
    2. [8.4.2 Users signup failure should not make a new user](#)
    3. [8.4.3 Users signup success should make a new user](#)
  9. [8.5 Conclusion](#)
  10. [8.6 Exercises](#)
17. [Chapter 9 Sign in, sign out](#)
18.
  1. [9.1 Sessions](#)
  2.
    1. [9.1.1 Sessions controller](#)
    2. [9.1.2 Signin form](#)
  3. [9.2 Signin failure](#)
  4.
    1. [9.2.1 Reviewing form submission](#)
    2. [9.2.2 Failed signin \(test and code\)](#)
  5. [9.3 Signin success](#)
  6.
    1. [9.3.1 The completed create action](#)
    2. [9.3.2 Remember me](#)
    3. [9.3.3 Current user](#)
  7. [9.4 Signing out](#)
  8.
    1. [9.4.1 Destroying sessions](#)
    2. [9.4.2 Signin upon signup](#)
    3. [9.4.3 Changing the layout links](#)
    4. [9.4.4 Signin/out integration tests](#)
  9. [9.5 Conclusion](#)
  10. [9.6 Exercises](#)
19. [Chapter 10 Updating, showing, and deleting users](#)
20.
  1. [10.1 Updating users](#)
  2.
    1. [10.1.1 Edit form](#)
    2. [10.1.2 Enabling edits](#)
  3. [10.2 Protecting pages](#)
  4.
    1. [10.2.1 Requiring signed-in users](#)
    2. [10.2.2 Requiring the right user](#)
    3. [10.2.3 Friendly forwarding](#)
  5. [10.3 Showing users](#)
  6.
    1. [10.3.1 User index](#)
    2. [10.3.2 Sample users](#)
    3. [10.3.3 Pagination](#)
    4.
      1. [Testing pagination](#)
    5. [10.3.4 Partial refactoring](#)
  7. [10.4 Destroying users](#)
  8.
    1. [10.4.1 Administrative users](#)
    2.
      1. [Revisiting attr\\_accessible](#)
    3. [10.4.2 The destroy action](#)
  9. [10.5 Conclusion](#)
  10. [10.6 Exercises](#)
21. [Chapter 11 User microposts](#)
22.
  1. [11.1 A Micropost model](#)
  2.
    1. [11.1.1 The basic model](#)
    2.
      1. [Accessible attribute](#)

3. [11.1.2 User/Micropost associations](#)
4. [11.1.3 Micropost refinements](#)
5.
  1. [Default scope](#)
  2. [Dependent: destroy](#)
6. [11.1.4 Micropost validations](#)
3. [11.2 Showing microposts](#)
4.
  1. [11.2.1 Augmenting the user show page](#)
  2. [11.2.2 Sample microposts](#)
5. [11.3 Manipulating microposts](#)
6.
  1. [11.3.1 Access control](#)
  2. [11.3.2 Creating microposts](#)
  3. [11.3.3 A proto-feed](#)
  4. [11.3.4 Destroying microposts](#)
  5. [11.3.5 Testing the new home page](#)
7. [11.4 Conclusion](#)
8. [11.5 Exercises](#)
23. [Chapter 12 Following users](#)
24.
  1. [12.1 The Relationship model](#)
  2.
    1. [12.1.1 A problem with the data model \(and a solution\)](#)
    2. [12.1.2 User/relationship associations](#)
    3. [12.1.3 Validations](#)
    4. [12.1.4 Following](#)
    5. [12.1.5 Followers](#)
  3. [12.2 A web interface for following and followers](#)
  4.
    1. [12.2.1 Sample following data](#)
    2. [12.2.2 Stats and a follow form](#)
    3. [12.2.3 Following and followers pages](#)
    4. [12.2.4 A working follow button the standard way](#)
    5. [12.2.5 A working follow button with Ajax](#)
  5. [12.3 The status feed](#)
  6.
    1. [12.3.1 Motivation and strategy](#)
    2. [12.3.2 A first feed implementation](#)
    3. [12.3.3 Scopes, subselects, and a lambda](#)
    4. [12.3.4 The new status feed](#)
  7. [12.4 Conclusion](#)
  8.
    1. [12.4.1 Extensions to the sample application](#)
    2.
      1. [Replies](#)
      2. [Messaging](#)
      3. [Follower notifications](#)
      4. [Password reminders](#)
      5. [Signup confirmation](#)
      6. [RSS feed](#)
      7. [REST API](#)
      8. [Search](#)
    3. [12.4.2 Guide to further resources](#)
  9. [12.5 Exercises](#)
  25. [Chapter 13 Rails 3.1](#)
  26.
    1. [13.1 Upgrading the sample app](#)
    2.
      1. [13.1.1 Installing and configuring Rails 3.1](#)
      2. [13.1.2 Getting to Red](#)
      3. [13.1.3 Minor issues](#)

4.
  1. [will paginate](#)
  2. [Pagination spec](#)
5. [13.1.4 Major differences](#)
6.
  1. [Asset directories](#)
  2. [Prototype to jQuery](#)
3. [13.2 New features in Rails 3.1](#)
4.
  1. [13.2.1 Asset pipeline](#)
  2. [13.2.2 Reversible migrations](#)
  3. [13.2.3 Sass and CoffeeScript](#)
5. [13.3 Exercises](#)

## 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 *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 *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

*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 *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 a programmer, educator, and entrepreneur. Michael was coauthor of *RailsSpace*, a best-selling Rails tutorial book published in 2007, and was cofounder and lead developer of [Insoshi](#), a popular social networking platform in Ruby on Rails. Previously, he taught theoretical and computational physics at the [California Institute of](#)

[Technology](#) (Caltech), where he received the Lifetime Achievement Award for Excellence in Teaching. Michael 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 Rails by Example*. Copyright © 2010 by Michael Hartl. All source code in *Ruby on Rails Tutorial* is available under the [MIT License](#) and the [Beerware License](#).

Copyright (c) 2010 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 BEERWARE 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 someday, 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.

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 -T
$ cd sample_app
```

Here the `-T` 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. (Recall that you might need version 1.2.5 of the `sqlite3` gem, depending on your system.) 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 10.42](#) at this time.)

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

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

gem 'rails', '3.0.9'
gem 'sqlite3', '1.3.3'

group :development do
  gem 'rspec-rails', '2.6.1'
end

group :test do
  gem 'rspec-rails', '2.6.1'
  gem 'webrat', '0.7.1'
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 also include a gem for [Webrat](#), a testing utility which used to be installed automatically as a dependency but now needs to be included explicitly.) To install and include the *RSpec* gems, we use `bundle install` as usual:

```
$ bundle install
```

In order to get Rails to use *RSpec* in place of `Test::Unit`, we need to install the files needed by *RSpec*. This can be accomplished with `rails generate`:

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

With that, all we have left is to initialize the Git repository:<sup>1</sup>

```
$ 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 and commit the changes:

```
$ git mv README README.markdown
$ git commit -a -m "Improved the README"
```

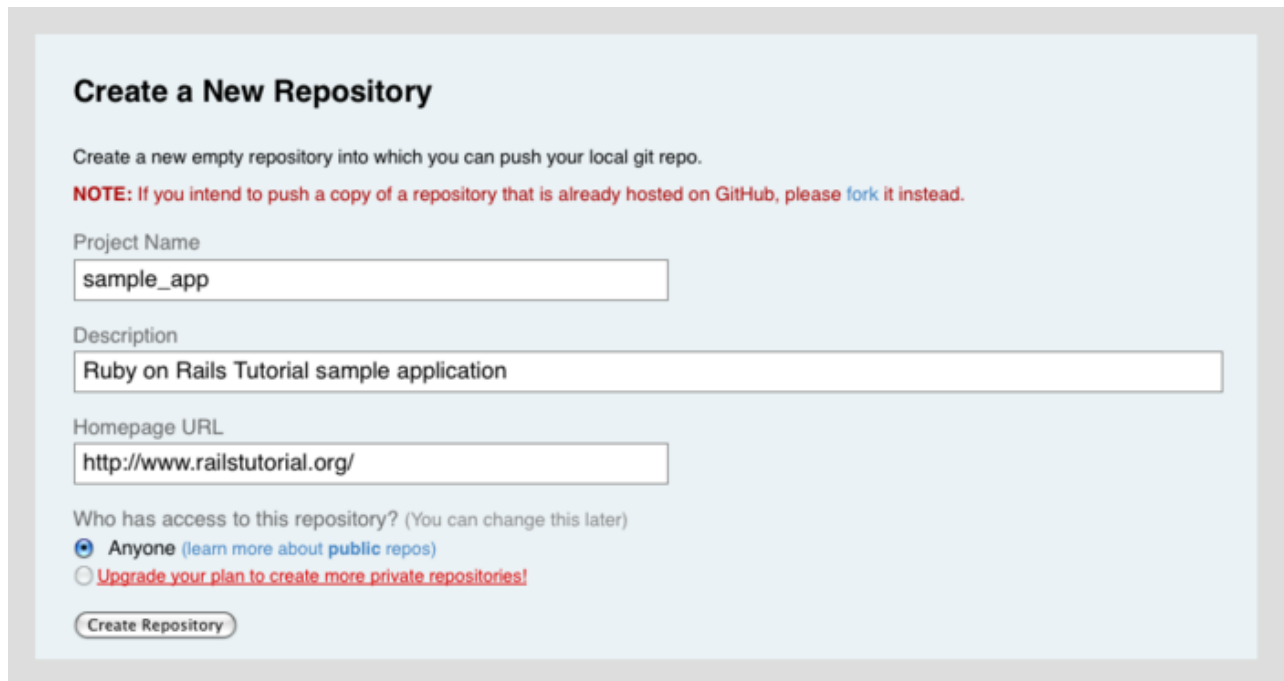


Figure 3.1: Creating the sample app repository at GitHub. [\(full size\)](#)

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 origin master
```

(Note that, as a result of this step, the repository at [http://github.com/railstutorial/sample\\_app](http://github.com/railstutorial/sample_app) has the source code for the full sample application. You are welcome to consult it for reference, with two caveats: (1) You will learn a lot more if you type in the source code samples yourself, rather than relying on the completed version; (2) There may be minor differences between the GitHub repository and the code in the book. This is due both to the incorporation of some of the book's exercises and to the repository's use in the [Rails Tutorial screencasts](#), which includes a few more tests.)

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

```
$ heroku create
$ git push heroku master
```

(If this doesn't work for you, see the note just above [Listing 1.9](#) for a possible fix.) As you proceed through the rest of

the book, I recommend pushing and deploying the application regularly:

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

With that, we're ready to get started developing the sample application.

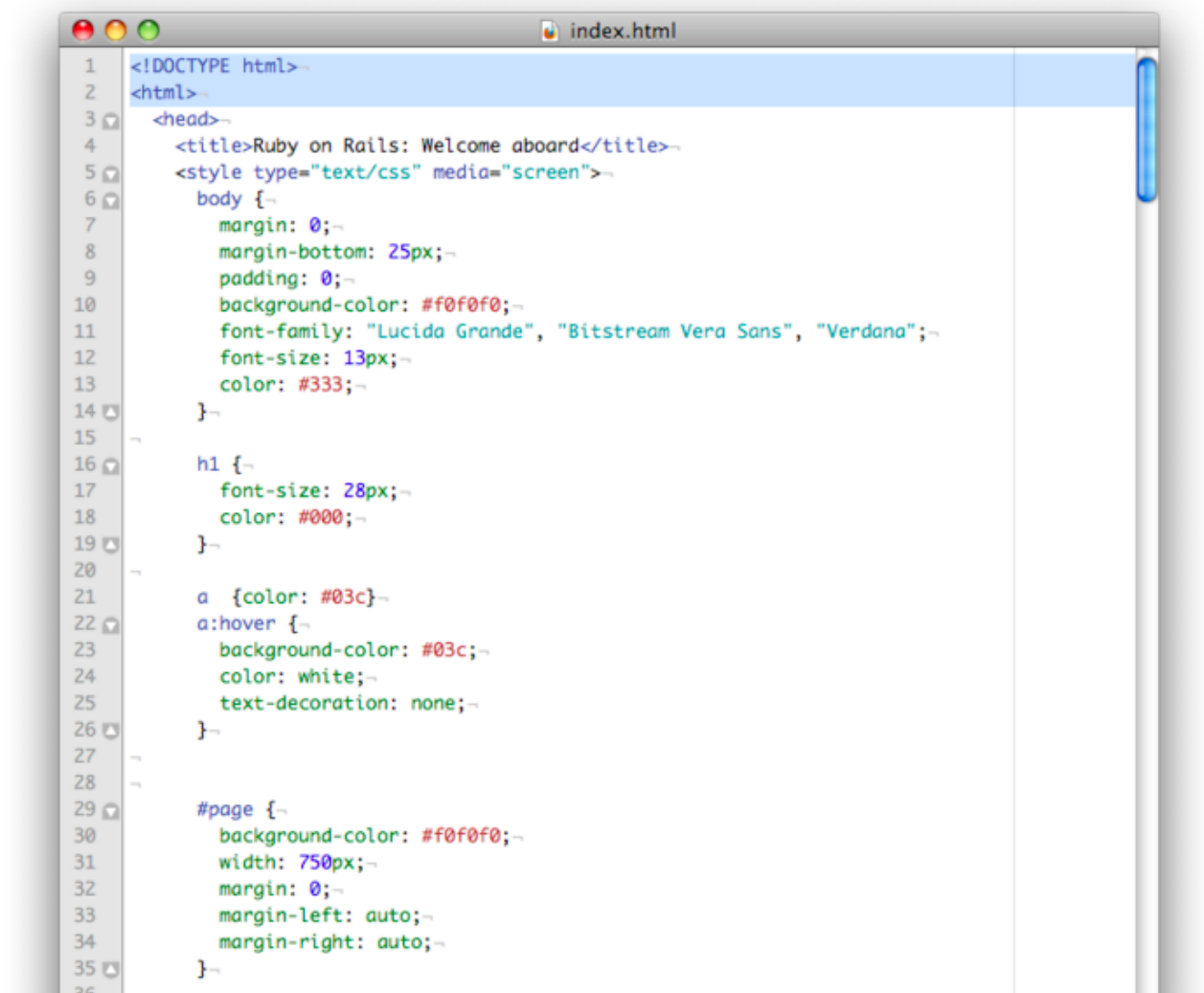
## 3.1 Static pages

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.)

### 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](#)).



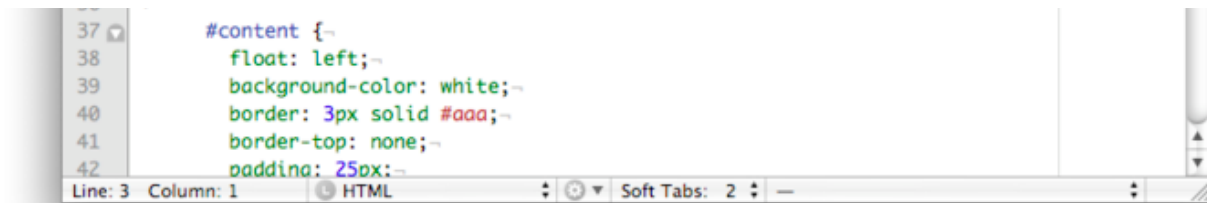


Figure 3.2: The `public/index.html` file. [\(full size\)](#)

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.<sup>2</sup> In the case of the special `index.html` file, you don't even have to indicate the file in the URL, 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](#)):<sup>3</sup>

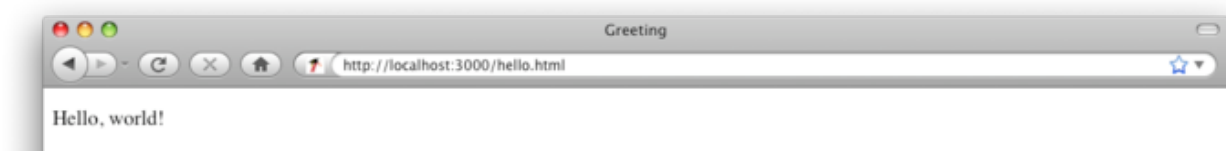
```
$ mate public/hello.html
```

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](#));<sup>4</sup> 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.) As promised, when visiting the address <http://localhost:3000/hello.html>, Rails renders it 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".



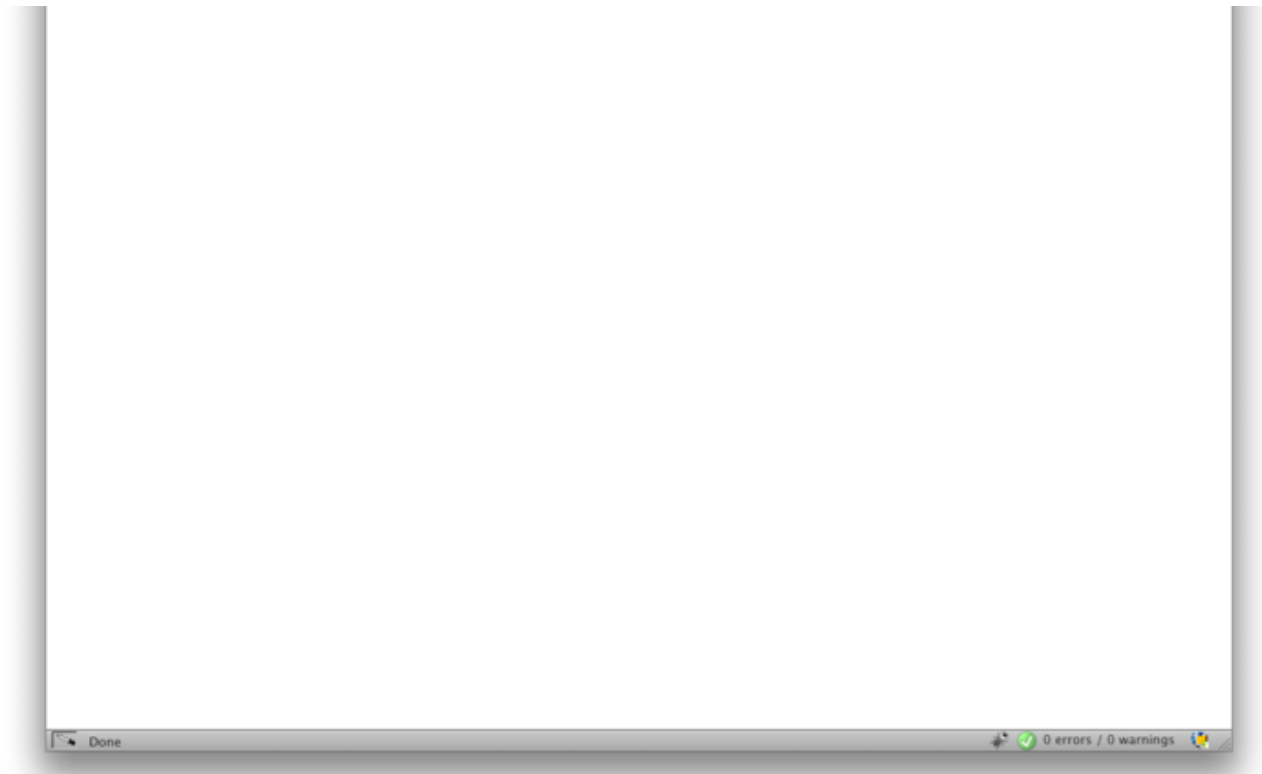


Figure 3.3: Our very own static HTML file (<http://localhost:3000/hello.html>). ([full size](#))

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 5.2](#) 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 URLs than static files.<sup>5</sup> 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. Since we're making this controller to handle (mostly) static pages, we'll just call it the Pages controller, and plan to make actions for a Home page, a Contact page, and an About page. The `generate` script takes an optional list of actions, so we'll include some of our initial actions directly on the command line:

Listing 3.4. Generating a Pages controller.

```
$ rails generate controller Pages home contact
  create  app/controllers/pages_controller.rb
  route   get "pages/contact"
  route   get "pages/home"
  invoke  erb
  create  app/views/pages
  create  app/views/pages/home.html.erb
  create  app/views/pages/contact.html.erb
  invoke  rspec
  create  spec/controllers/pages_controller_spec.rb
  create  spec/views/pages
  create  spec/views/pages/home.html.erb_spec.rb
  create  spec/views/pages/contact.html.erb_spec.rb
  invoke  helper
  create  app/helpers/pages_helper.rb
  invoke  rspec
```

(Note that, because we installed RSpec with `rails generate rspec:install`, the controller generation automatically creates RSpec test files in the `spec/` directory.) Here, I've intentionally "forgotten" the about page so that we can see how to add it in by hand ([Section 3.2](#)).

The Pages controller generation in [Listing 3.4](#) automatically updates the *routes* file, called `config/routes.rb`, which Rails uses to find the correspondence between URLs 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.

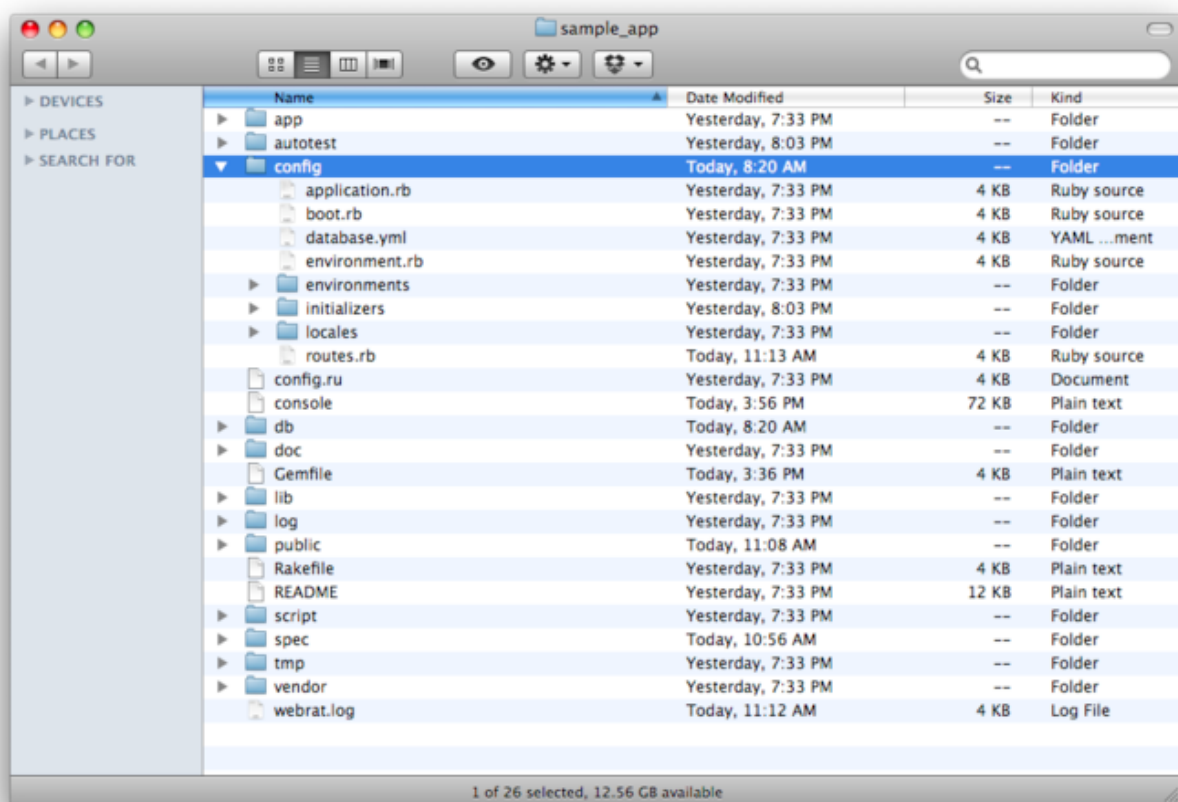


Figure 3.4: Contents of the sample app's `config` directory. [\(full size\)](#)

Since we generated `home` and `contact` 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 `contact` actions in the Pages controller.

`config/routes.rb`

```
SampleApp::Application.routes.draw do
  get "pages/home"
  get "pages/contact"
  .
  .
  .
end
```

Here the rule

```
get "pages/home"
```

maps requests for the URL `/pages/home` to the `home` action in the Pages 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 3.1](#)). In our case, this means that when we generate a `home` action inside the Pages controller we automatically get a page at the address `/pages/home`. To see the results, kill the server by hitting Ctrl-C, run `rails server`, and then navigate to [/pages/home](#) ([Figure 3.5](#)).

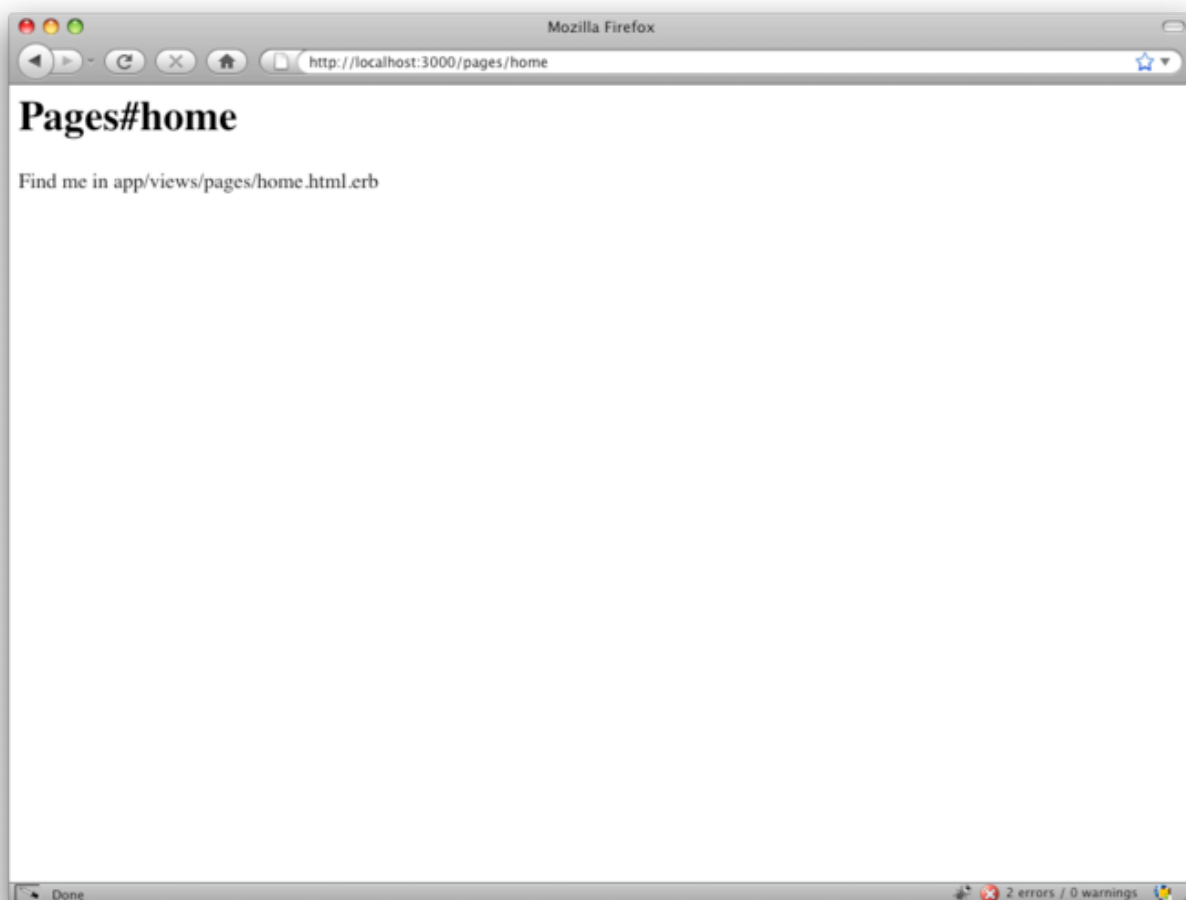


Figure 3.5: The raw home view (</pages/home>) generated by Rails. [\(full size\)](#)

Box 3.1.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 8](#).

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 craigslist.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 Pages 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 Pages controller does not follow the REST conventions.)

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

app/controllers/pages\_controller.rb

```
class PagesController < ApplicationController

  def home
  end

  def contact
  end
end
```

We see here that `pages_controller.rb` defines a *class* called `PagesController`. Classes are simply a convenient way to organize *functions* (also called *methods*) like the `home` and `contact` actions, which are defined using the `def` keyword. The angle bracket `<` indicates that `PagesController` *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 Pages controller, both its methods are initially empty:

```
def home
end

def contact
end
```

In plain Ruby, these methods would simply do nothing. In Rails, the situation is different; `PagesController` is a Ruby class, but because it inherits from `ApplicationController` the behavior of its methods is specific to Rails: when visiting the URL `/pages/home`, Rails looks in the Pages 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 hitting `/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/pages/home.html.erb
```

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

The view for the `contact` action is analogous ([Listing 3.8](#)).

Listing 3.8. The generated view for the Contact page.

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

```
<h1>Pages#contact</h1>
<p>Find me in app/views/pages/contact.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 first add the `about` action we “forgot” in [Section 3.1.2](#), add a very small amount of dynamic content, and then take the first steps toward styling the pages with CSS. Before moving on, if you're using Git it's a good idea to add the files for the Pages controller to the repository at this time:

```
$ git add .
$ git commit -m "Added a Pages controller"
```

## [3.2 Our first tests](#)

If you ask five Rails developers how to test any given piece of code, you'll get about fifteen different answers—but they'll all agree that you should definitely be writing tests. It's in this spirit that we'll approach testing our sample application, writing solid tests without worrying too much about making them perfect. You shouldn't take the tests in *Rails Tutorial* as gospel; they are based on the style I have developed during my own work and from reading the code of others. As you gain experience as a Rails developer, you will no doubt form your own preferences and develop your own testing style.

In addition to writing tests throughout the development of the sample application, we will also make the increasingly common choice about *when* to write tests by writing them *before* the application code—an approach known as *test-driven development*, or TDD.<sup>6</sup> Our specific example will be to add an About page to our sample site. Fortunately, adding the extra page is not hard—you might even be able to guess the answer based on the examples in the previous section—which means that we can focus on testing, which contains quite a few new ideas.

At first, testing for the existence of a page might seem like overkill, but experience shows that it is not. So many things can go wrong when writing software that having a good test suite is invaluable to assure quality. Moreover, it is common for computer programs—and especially web applications—to be constantly extended, and any time you make a change you risk introducing errors. Writing tests doesn't guarantee that these bugs won't happen, but it makes them much more likely to be caught (and fixed) when they occur. Furthermore, by writing tests for bugs that *do* happen, we can make them much less likely to recur.

(As noted in [Section 1.1.1](#), if you find the tests overwhelming, go ahead and skip them on first reading. Once you have a stronger grasp of Rails and Ruby, you can loop back and learn testing on a second pass.)

### 3.2.1 Testing tools

To write tests for our sample application, our main tool is a framework called [RSpec](#), which is a [domain-specific language](#) for describing the behavior of code, together with a program (called `rspec`) to verify the desired behavior. Designed for testing any Ruby program, RSpec has gained significant traction in the Rails community. Obie Fernandez, author of [The Rails 3 Way](#), has called RSpec “the Rails Way”, and I agree.<sup>7</sup>

If you followed the steps in the introduction, RSpec has already been installed via the Bundler Gemfile ([Listing 3.1](#)) and `bundle install`.

#### [Autotest](#)

Autotest is a tool that continuously runs your test suite in the background based on the specific file changes you make. For example, if you change a controller file, Autotest runs the tests for that controller. The result is instant feedback on the status of your tests. We’ll learn more about Autotest when we see it in action ([Section 3.2.2](#)).

Installing Autotest is optional, and configuring it can be a bit tricky, but if you can get it to work on your system I’m sure you’ll find it as useful as I do. To install Autotest, install the `autotest` and `autotest-rails-pure`<sup>8</sup> gems as follows:<sup>9</sup>

```
$ gem install autotest -v 4.4.6
$ gem install autotest-rails-pure -v 4.1.2
```

(If you get a permissions error here, recall from [Section 1.1.3](#) that you may have to use `sudo`.)

The next steps depend on your platform. I’ll go through the steps for OS X, since that’s what I use, and then give references to blog posts that discuss Autotest on Linux and Windows. On OS X, you should [install Growl](#) (if you don’t have it already) and then install the `autotest-fsevent` and `autotest-growl` gems:<sup>10</sup>

```
$ gem install autotest-fsevent -v 0.2.4
$ gem install autotest-growl -v 0.2.9
```

If FSEvent won’t install properly, double-check that [Xcode](#) is installed on your system.

To use the Growl and FSEvent gems, make an Autotest configuration file in your Rails root directory and fill it with the contents of [Listing 3.9](#) (or [Listing 3.10](#) if [Listing 3.9](#) gives an error on your system):

```
$ mate .autotest
```

Listing 3.9. The `.autotest` configuration file for Autotest on OS X.

```
require 'autotest/growl'
require 'autotest/fsevent'
```

Listing 3.10. An alternate `.autotest` file needed on some systems.

```
require 'autotest-growl'
require 'autotest-fsevent'
```

(Note: this will create an Autotest configuration for the sample application only; if you want to share this Autotest configuration with other Rails or Ruby projects, you should create the `.autotest` file in your *home* directory instead:

```
$ mate ~/.autotest
```

where ~ (tilde) is the Unix symbol for “[home directory](#)”).

If you’re running Linux with the Gnome desktop, you should try the steps at [Automate Everything](#), which sets up on Linux a system similar to Growl notifications on OS X. Windows users should try installing [Growl for Windows](#) and then follow the instructions at the [GitHub page for autotest-growl](#). Both Linux and Windows users might want to take a look at [autotest-notification](#); *Rails Tutorial* reader Fred Schoeneman has a write-up [about Autotest notification on his blog](#).<sup>11</sup>

### 3.2.2 TDD: Red, Green, Refactor

In test-driven development, we first write a *failing* test: in our case, a piece of code that expresses the idea that there “should be an about” page. Then we get the test to pass, in our case by adding the about action and corresponding view. The reason we don’t typically do the reverse—implement first, then test—is to make sure that we actually test for the feature we’re adding. Before I started using TDD, I was amazed to discover how often my “tests” actually tested the wrong thing, or even tested nothing at all. By making sure that the test fails first and *then* passes, we can be more confident that the test is doing the right thing.

It’s important to understand that TDD is not always the right tool for the job. In particular, 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.

One way to proceed in test-driven development is a cycle known as “Red, Green, Refactor”. The first step, Red, refers to writing a failing test, which many test tools indicate with the color red. The next step, Green, refers to a passing test, indicated with the color (wait for it) green. Once we have a passing test (or set of tests), we are free to *refactor* our code, changing the form (eliminating duplication, for example) without changing the function.

We don’t have any colors yet, so let’s get started toward Red. RSpec (and testing in general) can be a little intimidating at first, so we’ll use the tests generated by `rails generate controller Pages` in [Listing 3.4](#) to get us started. Since I’m not partial to separate tests for views or helpers, which I’ve found to be either brittle or redundant, our first step is to remove them. If you’re using Git, you can do this as follows:

```
$ git rm -r spec/views
$ git rm -r spec/helpers
```

Otherwise, remove them directly:

```
$ rm -rf spec/views
$ rm -rf spec/helpers
```

We’ll handle tests for views and helpers directly in the controller tests starting in [Section 3.3](#).

To get started with RSpec, take a look at the Pages controller spec<sup>12</sup> we just generated ([Listing 3.11](#)).

Listing 3.11. The generated Pages controller spec.

```
spec/controllers/pages_controller_spec.rb
```

```
require 'spec_helper'

describe PagesController do

  describe "GET 'home'" do
    it "should be successful" do
      get 'home'
```

```

        response.should be_success
      end
    end

    describe "GET 'contact'" do
      it "should be successful" do
        get 'contact'
        response.should be_success
      end
    end
  end
end

```

This code 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 is 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 3.11](#) contains two `describe` blocks, each with one *example* (i.e., a block starting with `it "..." do`). Let's focus on the first one to get a sense of what it does:

```

describe "GET 'home'" do
  it "should be successful" do
    get 'home'
    response.should be_success
  end
end

```

The first line indicates that we are describing a `GET` operation for the `home` action. This is just a description, and it can be anything you want; RSpec doesn't care, but you and other human readers probably do. In this case, `"GET 'home' "` indicates that the test corresponds to an HTTP `GET` request, as discussed in [Box 3.1](#). Then the spec says that when you visit the home page, it should be successful. As with the first line, what goes inside the quote marks is irrelevant to RSpec, and is intended to be descriptive to human readers. The third line, `get 'home'`, is the first line that really does something. Inside of RSpec, this line *actually submits a GET request*; in other words, it acts like a browser and hits a page, in this case `/pages/home`. (It knows to hit the Pages controller automatically because this is a Pages controller test; it knows to hit the home page because we tell it to explicitly.) Finally, the fourth line says that the *response* of our application should indicate success (i.e., it should return a *status code* of 200; see [Box 3.2](#)).

### Box 3.2.HTTP response codes

After a client (such as a web browser) sends a request corresponding to one of the HTTP verbs ([Box 3.1](#)), the web server *responds* with a numerical code indicating the [HTTP status](#) of the response. For example, a status code of 200 means “success”, and a status code of 301 means “permanent redirect”. If you [install curl](#), a command-line client that can issue HTTP requests, you can see this directly at, e.g., `www.google.com` (where the `--head` flag prevents `curl` from returning the whole page):

```

$ curl --head www.google.com
HTTP/1.1 200 OK
.
.
.

```

Here Google indicates that the request was successful by returning the status `200 OK`. In contrast, `google.com` is permanently redirected (to `www.google.com`, naturally), indicated by status code 301 (a “301 redirect”):

```
$ curl --head google.com
HTTP/1.1 301 Moved Permanently
Location: http://www.google.com/
.
.
.
```

(Note: The above results may vary by country.)

When we write `response.should be_success` in an RSpec test, RSpec verifies that our application's response to the request is status code 200.

Now it's time to run our tests. There are several different and mostly equivalent ways to do this.<sup>[13](#)</sup> One way to run all the tests is to use the `rspec` script at the command line as follows:<sup>[14](#)</sup>

```
$ bundle exec rspec spec/
....
```

```
Finished in 0.07252 seconds
```

```
2 examples, 0 failures
```

(Unfortunately, lots of things can go wrong at this point. If the test suite fails, try migrating the database with `bundle exec rake db:migrate` as described in [Section 2.2](#). If RSpec doesn't work at all, try uninstalling and reinstalling it:

```
$ gem uninstall rspec rspec-rails
$ bundle install
```

If it still doesn't work and you're using [RVM](#), try removing the Rails Tutorial gemset and reinstalling the gems:

```
$ rvm gemset delete rails3tutorial
$ rvm --create use 1.9.2@rails3tutorial
$ rvm --default 1.9.2@rails3tutorial
$ gem install rails -v 3.0.9
$ bundle install
```

If it still doesn't work, I'm out of ideas.)

When running `bundle exec rspec spec/`, `rspec` is a program provided by RSpec, while `spec/` is the *directory* whose specs you want to run. You can also run only the specs in a particular subdirectory. For example, this command runs only the controller specs:

```
$ bundle exec rspec spec/controllers/
....
```

```
Finished in 0.07502 seconds
```

```
2 examples, 0 failures
```

You can also run a single file:

```
$ bundle exec rspec spec/controllers/pages_controller_spec.rb
....
```

```
Finished in 0.07253 seconds
```

```
2 examples, 0 failures
```

Note that, as in [Section 2.2](#), we have used `bundle exec` to use the executable (in this case, `rspec`) corresponding to the gems in our application's `Gemfile`. Since this construction is a bit verbose, Bundler allows the creation of the associated binaries as follows:

```
$ bundle install --binstubs
```

This 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
```

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.

The results of all three commands above are the same since the `Pages controller spec` is currently our only test file. Throughout the rest of this book, I won't usually show the output of running the tests, but you should run `bundle exec rspec spec/` (or one of its variants) regularly as you follow along—or, better yet, use `Autotest` to run the test suite automatically. Speaking of which...

If you've installed `Autotest`, you can run it on your `RSpec` tests using the `autotest` command:

```
$ autotest
```

If you're using a Mac with `Growl` notifications enabled, you might be able to replicate my setup, shown in [Figure 3.6](#). With `Autotest` running in the background and `Growl` notifications telling you the status of your tests, TDD can be positively addictive.



Figure 3.6: Autotest (via autotest) in action, with a Growl

notification. [\(full size\)](#)

## Spork

You may have noticed that the overhead involved in running a test suite can be considerable. This is because each time RSpec runs the tests it has to reload the entire Rails environment. The [Spork test server<sup>15</sup>](#) 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 Autotest.

Configuring Spork and getting it to work can be difficult, and this is a rather advanced topic. **If you get stuck, don't hesitate to skip this section for now.**

The first step is to add the spork gem dependency to the Gemfile ([Listing 3.12](#)).

Listing 3.12. A Gemfile for the sample app.

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

gem 'rails', '3.0.9'
gem 'sqlite3', '1.3.3'

group :development do
  gem 'rspec-rails', '2.6.1'
end

group :test do
  gem 'rspec-rails', '2.6.1'
  .
  .
  .
  gem 'spork', '0.9.0.rc8'
end
```

Then install it:

```
$ bundle install
```

Next, bootstrap the Spork configuration:

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

Now we need to edit the RSpec configuration file, `spec/spec_helper.rb`, so that the environment gets loaded in a *prefork* block, which arranges for it to be loaded only once ([Listing 3.13](#)). *Note:* Only use this code if you are also using Spork. If you try to use [Listing 3.13](#) without Spork, your application test suite will not run.

Listing 3.13. Adding environment loading to the Spork.prefork block.

```
spec/spec_helper.rb

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.
ENV["RAILS_ENV"] ||= 'test'
require File.expand_path("../../config/environment", __FILE__)
require 'rspec/rails'

# Requires supporting files with custom matchers and macros, etc,
# in ./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

  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, comment the following line or assign false
  # instead of true.
  config.use_transactional_fixtures = true
end
end

Spork.each_run do
end
```

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

```
$ time rspec spec/
..
```

```
Finished in 0.09606 seconds
2 examples, 0 failures
```

```
real    0m7.445s
user    0m5.248s
sys     0m1.475s
```

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 Rails root directory, and then start a Spork server:

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



In another terminal window, we can now run our test suite with the `--drb` option<sup>16</sup> and verify that the environment-loading overhead is greatly reduced:

```
$ time rspec --drb spec/
..
```

```
Finished in 0.10519 seconds
2 examples, 0 failures
```

```
real    0m0.803s
user    0m0.354s
sys     0m0.171s
```

As expected, the overhead has been dramatically reduced.

To run RSpec and Spork with Autotest, we need to configure RSpec to use the `--drb` option by default, which we can arrange by adding it to the `.rspec` configuration file in the Rails root directory ([Listing 3.14](#)).

Listing 3.14. Adding the `--drb` option to the `.rspec` file.

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

With this updated `.rspec` file, the test suite should run as quickly as before, even without the explicit `--drb` option:

```
$ time rspec spec/
..
```

```
Finished in 0.10926 seconds
2 examples, 0 failures
```

```
real    0m0.803s
user    0m0.355s
sys     0m0.171s
```

Of course, running `time` here is just for purposes of illustration; normally, you just run

```
$ bundle exec rspec spec/
```

or

```
$ autotest
```

without the `time` command.

One word of advice when using Spork: if your tests are failing when you think they should be passing, the problem might be the Spork prefork loading, which can sometimes prevent necessary files from being re-loaded. When in doubt, 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
```

## Red

Now let's 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 3.15](#)).

Listing 3.15. The Pages controller spec with a failing test for the About page.

spec/controllers/pages\_controller\_spec.rb

```
require 'spec_helper'

describe PagesController do
  render_views

  describe "GET 'home'" do
    it "should be successful" do
      get 'home'
      response.should be_success
    end
  end

  describe "GET 'contact'" do
    it "should be successful" do
      get 'contact'
      response.should be_success
    end
  end

  describe "GET 'about'" do
    it "should be successful" do
      get 'about'
      response.should be_success
    end
  end
end
```

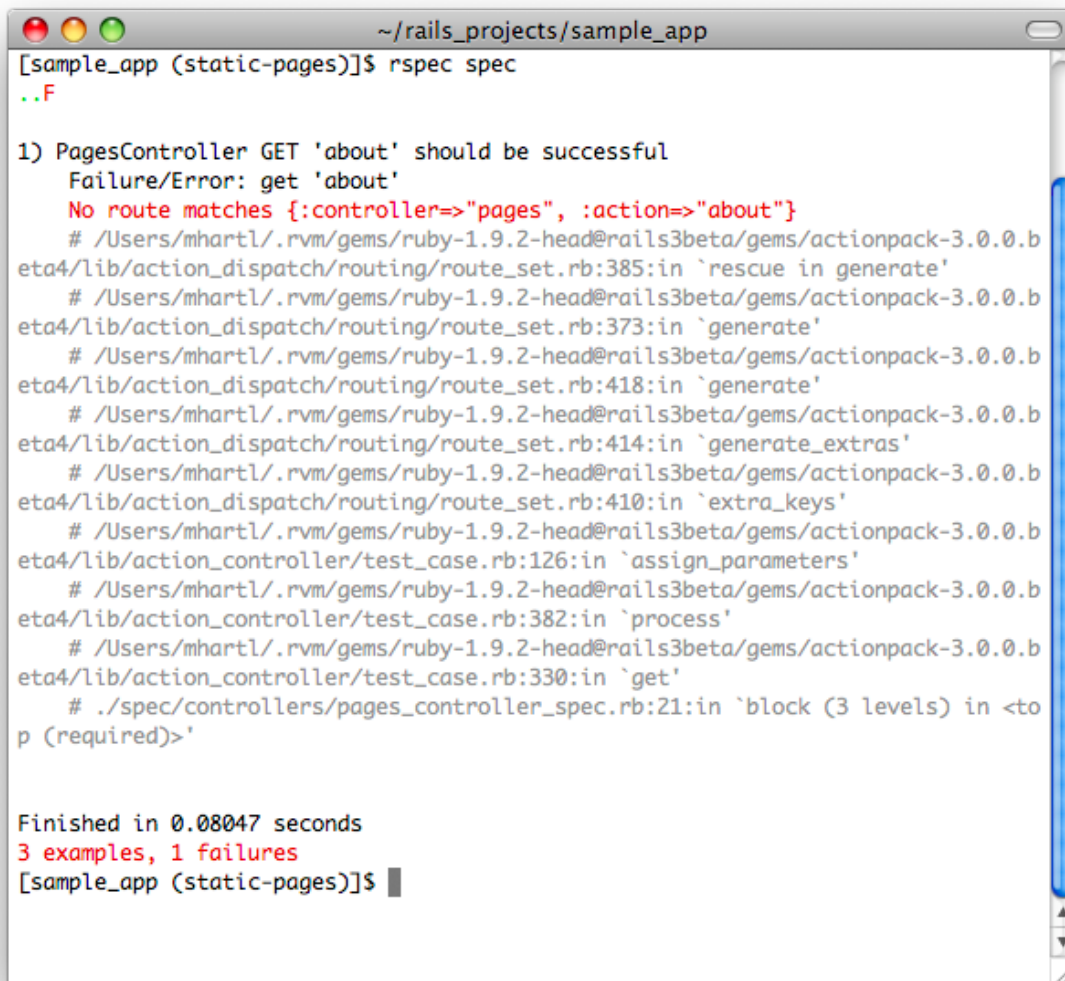
Note that we've added a line to tell RSpec to *render the views* inside the controller tests. In other words, by default RSpec just tests actions inside a controller test; if we want it also to render the views, we have to tell it explicitly via the second line:

```
describe PagesController do
  render_views
  .
  .
  .
```

This ensures that if the test passes, the page is really there.

The new test attempts to get the about action, and indicates that the resulting response should be a success. By design, it fails (with a red error message), as seen in [Figure 3.7](#) (`rspec spec/`) and [Figure 3.8](#) (`autotest`). (If you test the views in the controllers as recommended in this tutorial, it's worth noting that changing the view file won't prompt Autotest to run the corresponding controller test. There's probably a way to configure Autotest to do this automatically, but usually I just switch to the controller and press "space-backspace" so that the file gets marked as

modified. Saving the controller then causes Autotest to run the tests as desired.)

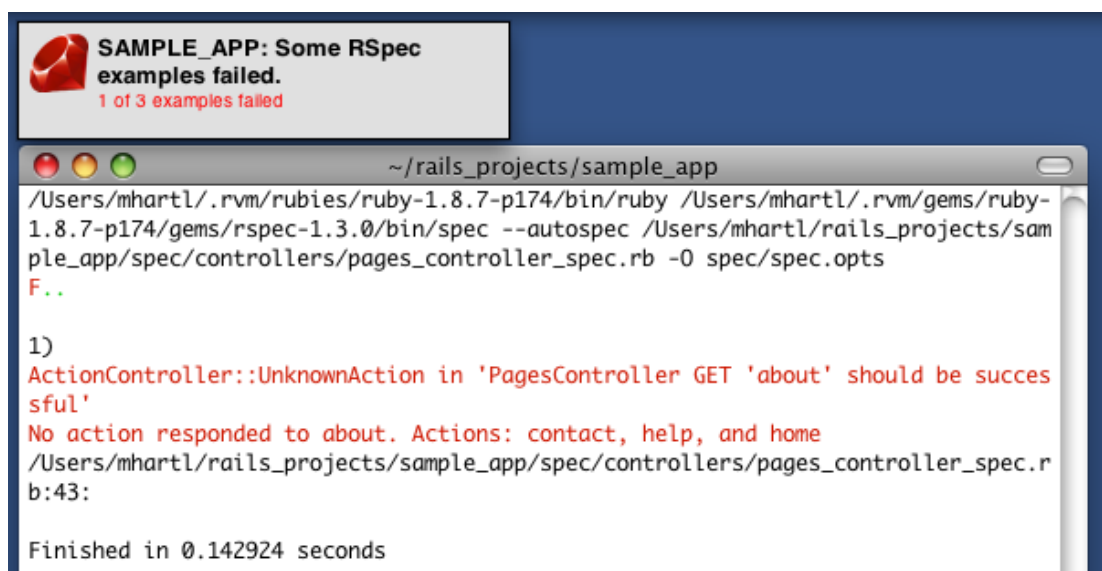


```
~/rails_projects/sample_app
[sample_app (static-pages)]$ rspec spec
..F

1) PagesController GET 'about' should be successful
   Failure/Error: get 'about'
     No route matches {:controller=>"pages", :action=>"about"}
     # /Users/mhartl/.rvm/gems/ruby-1.9.2-head@rails3beta/gems/actionpack-3.0.0.b
eta4/lib/action_dispatch/routing/route_set.rb:385:in `rescue in generate'
     # /Users/mhartl/.rvm/gems/ruby-1.9.2-head@rails3beta/gems/actionpack-3.0.0.b
eta4/lib/action_dispatch/routing/route_set.rb:373:in `generate'
     # /Users/mhartl/.rvm/gems/ruby-1.9.2-head@rails3beta/gems/actionpack-3.0.0.b
eta4/lib/action_dispatch/routing/route_set.rb:418:in `generate'
     # /Users/mhartl/.rvm/gems/ruby-1.9.2-head@rails3beta/gems/actionpack-3.0.0.b
eta4/lib/action_dispatch/routing/route_set.rb:414:in `generate_extras'
     # /Users/mhartl/.rvm/gems/ruby-1.9.2-head@rails3beta/gems/actionpack-3.0.0.b
eta4/lib/action_dispatch/routing/route_set.rb:410:in `extra_keys'
     # /Users/mhartl/.rvm/gems/ruby-1.9.2-head@rails3beta/gems/actionpack-3.0.0.b
eta4/lib/action_controller/test_case.rb:126:in `assign_parameters'
     # /Users/mhartl/.rvm/gems/ruby-1.9.2-head@rails3beta/gems/actionpack-3.0.0.b
eta4/lib/action_controller/test_case.rb:382:in `process'
     # /Users/mhartl/.rvm/gems/ruby-1.9.2-head@rails3beta/gems/actionpack-3.0.0.b
eta4/lib/action_controller/test_case.rb:330:in `get'
     # ./spec/controllers/pages_controller_spec.rb:21:in `block (3 levels) in <to
p (required)>'

Finished in 0.08047 seconds
3 examples, 1 failures
[sample_app (static-pages)]$
```

Figure 3.7: Failing spec for the About page using `rspec spec/`. [\(full size\)](#)



```
SAMPLE_APP: Some RSpec
examples failed.
1 of 3 examples failed

~/rails_projects/sample_app
/Users/mhartl/.rvm/rubies/ruby-1.8.7-p174/bin/ruby /Users/mhartl/.rvm/gems/ruby-
1.8.7-p174/gems/rspec-1.3.0/bin/spec --autospec /Users/mhartl/rails_projects/sam
ple_app/spec/controllers/pages_controller_spec.rb -0 spec/spec.opts
F..

1)
   ActionController::UnknownAction in 'PagesController GET 'about' should be succes
sful'
   No action responded to about. Actions: contact, help, and home
   /Users/mhartl/rails_projects/sample_app/spec/controllers/pages_controller_spec.r
b:43:

Finished in 0.142924 seconds
```

Figure 3.8:  
Failing  
spec for  
the About  
page using



Autotest. [\(full size\)](#)

This is Red. Now let's get to Green.

### [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 Pages 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.

Following the models provided by `home` and `contact` from [Listing 3.6](#), let's first add an `about` action in the Pages controller ([Listing 3.16](#)).

Listing 3.16. The Pages controller with added `about` action.

`app/controllers/pages_controller.rb`

```
class PagesController < ApplicationController

  def home
  end

  def contact
  end

  def about
  end
end
```

Next, we'll add the `about` action to the routes file ([Listing 3.17](#)).

Listing 3.17. Adding the `about` route.

`config/routes.rb`

```
SampleApp::Application.routes.draw do
  get "pages/home"
  get "pages/contact"
  get "pages/about"
  .
  .
  .
end
```

Finally, we'll add the `about` view. Eventually we'll fill it with something more informative, but for now we'll just mimic

the content from the generated views ([Listing 3.7](#) and [Listing 3.8](#)) for the about view ([Listing 3.18](#)).

Listing 3.18. A stub About page.

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

```
<h1>Pages#about</h1>
<p>Find me in app/views/pages/about.html.erb</p>
```

Running the specs or watching the update from Autotest ([Figure 3.9](#)) should get us back to Green:

```
$ bundle exec rspec spec/
```

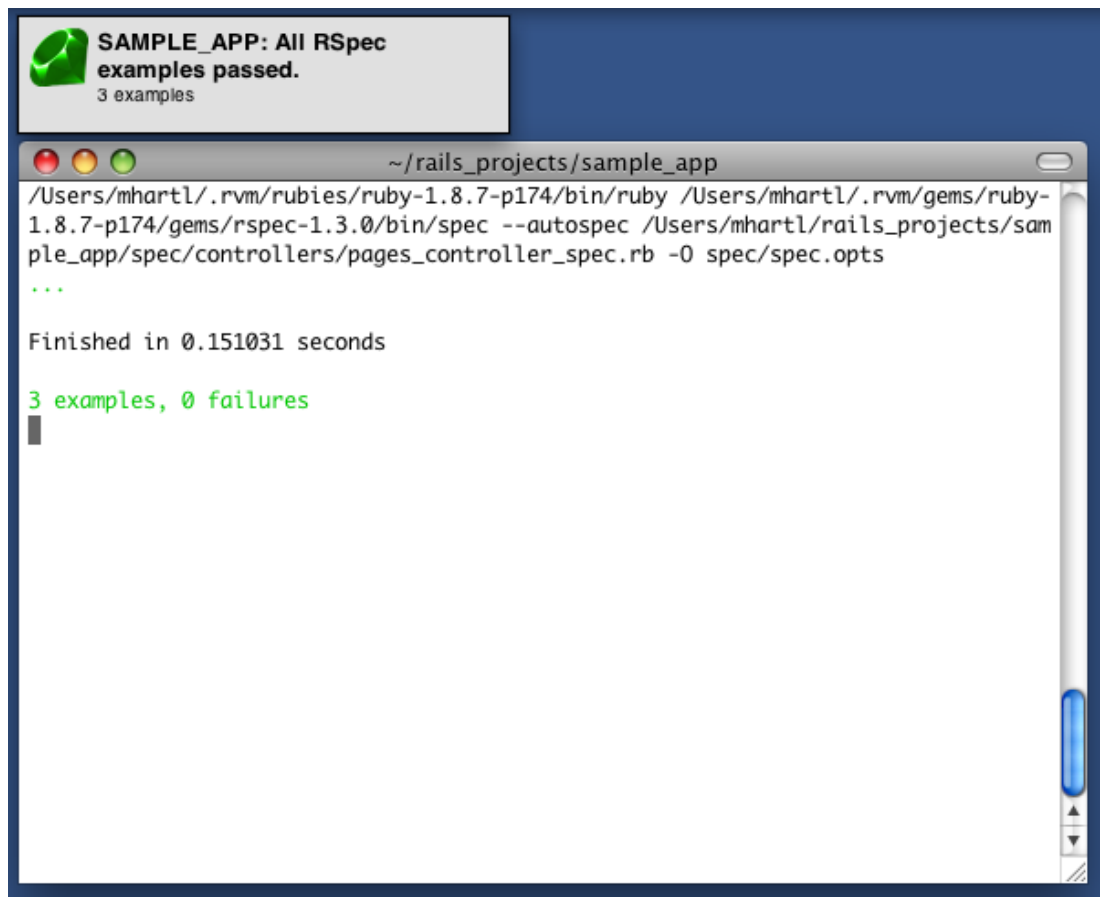


Figure 3.9:  
Autotest  
back to  
Green: All  
tests

passing. ([full size](#))

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 3.10](#)).





Figure 3.10: The new (and rather raw) About page (</pages/about>). [\(full size\)](#)

### [Refactor](#)

Now that we're at Green, we are free to *refactor* our code by changing its form without changing its function. 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 (passing!) 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 3.3.3](#) of this chapter.

## [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 this represents *truly* dynamic content is debatable, but in any case it lays the necessary foundation for unambiguously dynamic content in [Chapter 8](#).

(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 3.16](#), [Listing 3.17](#), and [Listing 3.18](#).)

### [3.3.1 Testing a title change](#)

Our plan is to edit the Home, Contact, and About pages to add the kind of HTML structure we saw in [Listing 3.3](#), including titles that change on each page. It's a delicate matter to decide just which of these changes to test, and in general testing HTML can be quite fragile since content tends to change frequently. We'll keep our tests simple by just testing for the page titles.

Page	URL	Base title	Variable title
Home	<code>/pages/home</code>	"Ruby on Rails Tutorial Sample App"	"   Home"
Contact	<code>/pages/contact</code>	"Ruby on Rails Tutorial Sample App"	"   Contact"

About /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 3.15](#), adding title tests following the model in [Listing 3.19](#).

Listing 3.19. A title test.

```
it "should have the right title" do
  get 'home'
  response.should have_selector("title",
    :content => "Ruby on Rails Tutorial Sample App | Home")
end
```

This uses the `have_selector` method inside RSpec; the documentation for `have_selector` is surprisingly sparse, but what it does is to check for an HTML element (the “selector”) with the given content. In other words, the code

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

checks to see that the content inside the `<title></title>` tags is “Ruby on Rails Tutorial Sample App | Home”.<sup>[17](#)</sup> It’s worth mentioning that the content need not be an exact match; any substring works as well, so that

```
response.should have_selector("title", :content => " | Home")
```

will also match the full title.<sup>[18](#)</sup>

Note that in [Listing 3.19](#) I’ve broken the material inside `have_selector` into two lines; this tells you something important about Ruby syntax: Ruby doesn’t care about newlines.<sup>[19](#)</sup> 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.<sup>[20](#)</sup> As it stands, I still find this code formatting rather ugly; [Section 3.5](#) has a refactoring exercise that makes them much prettier.<sup>[21](#)</sup>

Adding new tests for each of our three static pages following the model of [Listing 3.19](#) gives us our new Pages controller spec ([Listing 3.20](#)).

Listing 3.20. The Pages controller spec with title tests.

```
spec/controllers/pages_controller_spec.rb

require 'spec_helper'

describe PagesController do
  render_views

  describe "GET 'home'" do
    it "should be successful" do
      get 'home'
      response.should be_success
    end

    it "should have the right title" do
      get 'home'
      response.should have_selector("title",
        :content => "Ruby on Rails Tutorial Sample App | Home")
    end
  end
end
```

```
end

describe "GET 'contact'" do
  it "should be successful" do
    get 'contact'
    response.should be_success
  end

  it "should have the right title" do
    get 'contact'
    response.should have_selector("title",
                                  :content =>
                                    "Ruby on Rails Tutorial Sample App | Contact")
  end
end

describe "GET 'about'" do
  it "should be successful" do
    get 'about'
    response.should be_success
  end

  it "should have the right title" do
    get 'about'
    response.should have_selector("title",
                                  :content =>
                                    "Ruby on Rails Tutorial Sample App | About")
  end
end
end
```

Note that the `render_views` line introduced in [Listing 3.15](#) is necessary for the title tests to work.

With these tests in place, you should run

```
$ bundle exec rspec spec/
```

or use Autotest 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 3.21](#)), using the same basic HTML skeleton as in the "hello" page from [Listing 3.3](#).

*Note:* In Rails 3, the controller generator creates a *layout* file, whose purpose we will explain shortly, but which for now you should remove before proceeding:

```
$ rm app/views/layouts/application.html.erb
```

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

```
app/views/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 3.21](#) uses the title tested for in [Listing 3.20](#):

```
<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 Contact and About tests, and we can get to Green with the code in [Listing 3.22](#) and [Listing 3.23](#).

Listing 3.22. The view for the Contact page with full HTML structure.

app/views/pages/contact.html.erb

```
<!DOCTYPE html>
<html>
  <head>
    <title>Ruby on Rails Tutorial Sample App | Contact</title>
  </head>
  <body>
    <h1>Contact</h1>
    <p>
      Contact Ruby on Rails Tutorial about the sample app at the
      <a href="http://railstutorial.org/feedback">feedback page</a>.
    </p>
  </body>
</html>
```

Listing 3.23. The view for the About page with full HTML structure.

app/views/pages/about.html.erb

```
<!DOCTYPE html>
<html>
  <head>
    <title>Ruby on Rails Tutorial Sample App | About</title>
  </head>
  <body>
    <h1>About Us</h1>
    <p>
      <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>
```

These example pages introduce the *anchor* tag `a`, which creates links to the given URL (called an “href”, or “hypertext reference”, in the context of an anchor tag):

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

You can see the results in [Figure 3.11](#).

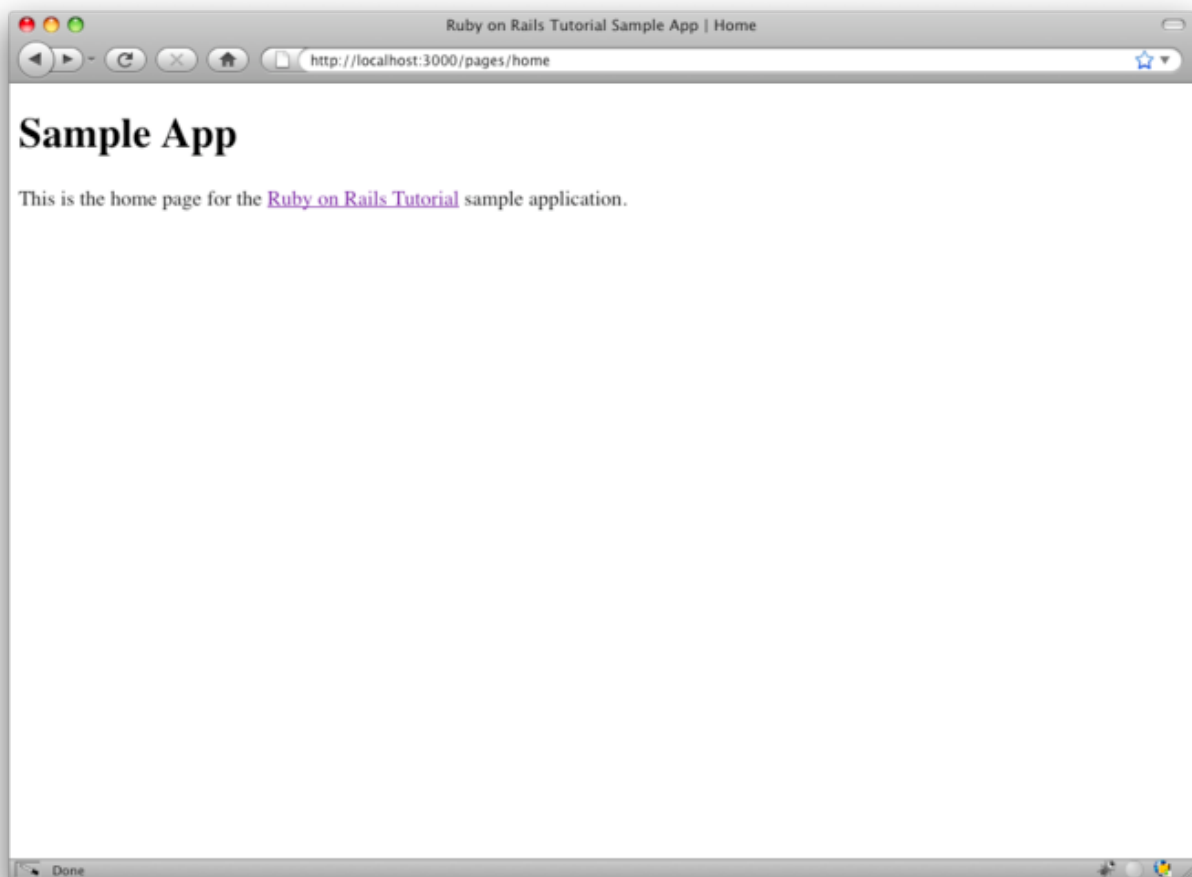


Figure 3.11: A minimal Home page for the sample app (</pages/home>). [\(full size\)](#)

### [3.3.3 Instance variables and 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 creating *instance variables* inside our actions. Since the Home, Contact, and About page titles have a variable component, we'll set the variable `@title` (pronounced "at title") to the appropriate title for each action ([Listing 3.24](#)).

Listing 3.24. The Pages controller with per-page titles.

app/controllers/pages\_controller.rb

```
class PagesController < ApplicationController

  def home
    @title = "Home"
  end

  def contact
    @title = "Contact"
  end

  def about
    @title = "About"
  end
end
```

A statement such as

```
@title = "Home"
```

is an *assignment*, in this case creating a new variable `@title` with value "Home". The at sign `@` in `@title` indicates that it is an instance variable. Instance variables have a more general meaning in Ruby (see [Section 4.2.3](#)), but in Rails their role is primarily to link actions and views: any instance variable defined in the `home` action is automatically available in the `home.html.erb` view, and so on for other action/view pairs.<sup>22</sup>

We can see how this works by replacing the literal title "Home" with the contents of the `@title` variable in the `home.html.erb` view ([Listing 3.25](#)).

Listing 3.25. The view for the Home page with an Embedded Ruby title.

app/views/pages/home.html.erb

```
<!DOCTYPE html>
<html>
  <head>
    <title>Ruby on Rails Tutorial Sample App | <%= @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 3.25](#) 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 mechanism in Rails for including dynamic content in web pages.<sup>23</sup> The code

```
<%= @title %>
```

indicates using `<%= ... %>` that Rails should insert the contents of the `@title` variable, whatever it may be. When we visit `/pages/home`, Rails executes the body of the `home` action, which makes the assignment `@title = "Home"`, so in the present case

```
<%= @title %>
```

gets replaced with “Home”. Rails then renders the view, using ERb to insert the value of `@title` into the template, which the web server then sends to your browser as HTML. The result 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. Then we can make the corresponding replacements for the Contact and About pages ([Listing 3.26](#) and [Listing 3.27](#)).

Listing 3.26. The view for the Contact page with an Embedded Ruby title.

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

```
<!DOCTYPE html>
<html>
  <head>
    <title>Ruby on Rails Tutorial Sample App | <%= @title %></title>
  </head>
  <body>
    <h1>Contact</h1>
    <p>
      Contact Ruby on Rails Tutorial about the sample app at the
      <a href="http://railstutorial.org/feedback">feedback page</a>.
    </p>
  </body>
</html>
```

Listing 3.27. The view for the About page with an Embedded Ruby title.

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

```
<!DOCTYPE html>
<html>
  <head>
    <title>Ruby on Rails Tutorial Sample App | <%= @title %></title>
  </head>
  <body>
    <h1>About Us</h1>
    <p>
      <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>
```

As before, the tests still pass.

### 3.3.4 Eliminating duplication with layouts

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

```
<!DOCTYPE html>
<html>
  <head>
    <title>Ruby on Rails Tutorial Sample App | <%= @title %></title>
  </head>
  <body>
    Contents
  </body>
</html>
```

In other words, *all* our pages are identical in structure, including even the title (because of Embedded Ruby), with the sole exception of the contents of each page.

Wouldn't it be nice if there were a way to factor out the common elements into some sort of global layout, with the body contents inserted on a per-page basis? Indeed, it would be nice, and Rails happily obliges using a special file called `application.html.erb`, which lives in the `layouts` directory. To capture the structural skeleton, create the file `application.html.erb` and fill it with the contents of [Listing 3.28](#).

Listing 3.28. The sample application site layout.

`app/views/layouts/application.html.erb`

```
<!DOCTYPE html>
<html>
  <head>
    <title>Ruby on Rails Tutorial Sample App | <%= @title %></title>
    <%= csrf_meta_tag %>
  </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. As with `<%= @title %>`, the `<%= ... %>` tags indicate Embedded Ruby, and the equals sign in `<%= ... %>` ensures that the results of evaluating the expression are inserted at that exact point in the template. (Don't worry about the meaning of the word "yield" in this context;<sup>[24](#)</sup> what matters is that using this layout ensures that visiting the page `/pages/home` converts the contents of `home.html.erb` to HTML and then inserts it in place of `<%= yield %>`).

Now that we have a site-wide layout, we've also taken this opportunity to add a security feature to each page.

[Listing 3.28](#) adds the code

```
<%= csrf_meta_tag %>
```

which uses the Rails method `csrf_meta_tag` to prevent [cross-site request forgery](#) (CSRF), a type of malicious web attack. Don't worry about the details (I don't); just know that Rails is working hard to keep your application secure.

Of course, the views in [Listing 3.25](#), [Listing 3.26](#), and [Listing 3.27](#) are still filled with all the HTML structure we just hoisted into the layout, so we have to rip it out, leaving only the interior contents. The resulting cleaned-up views appear in [Listing 3.29](#), [Listing 3.30](#), and [Listing 3.31](#).

Listing 3.29. The Home view with HTML structure removed.

```
app/views/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>
```

Listing 3.30. The Contact view with HTML structure removed.

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

```
<h1>Contact</h1>
<p>
  Contact Ruby on Rails Tutorial about the sample app at the
  <a href="http://railstutorial.org/feedback">feedback page</a>.
</p>
```

Listing 3.31. The About view with HTML structure removed.

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

```
<h1>About Us</h1>
<p>
  <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, Contact, and About pages are exactly the same as before—i.e., we have successfully refactored them—but they have much less duplication. And, as required, the tests still pass.

## 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 "Done with 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):

```
$ bundle exec rspec spec/
$ git push
```

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

```
$ bundle exec rspec spec/
$ git push heroku
```

Note that in both cases I've run `rspec spec/`, just to be sure that all the tests still pass. Running your tests before pushing or deploying is a good habit to cultivate.

## 3.5 Exercises

1. Make a Help page for the sample app. First write a test for the existence of a page at the URL `/pages/help`. Then write a second test for the title “Ruby on Rails Tutorial Sample App | Help”. Get your tests to pass, and then fill in the Help page with the content from [Listing 3.32](#).
2. You may have noticed some repetition in the Pages controller spec ([Listing 3.20](#)). In particular, the base title, “Ruby on Rails Tutorial Sample App”, is the same for every title test. Using the RSpec `before(:each)` facility, which executes a block of code before each test case, fill in [Listing 3.33](#) to define a `@base_title` instance variable that eliminates this duplication. (This code uses two new elements: a *symbol*, `:each`, and the string concatenation operator `+`. We'll learn more about both in [Chapter 4](#), and we'll see `before(:each)` again in [Section 6.2.1](#).) Note that, with the base title captured in an instance variable, we are now able to align `:content` with the first character inside each left parenthesis (`.`). This is my preferred convention for formatting code broken into multiple lines.
3. The Autotest gem installations in [Section 3.2.1.1](#) should probably be in the `:test` area of Gemfile. Add Autotest to the Gemfile by uncommenting the appropriate lines in [Listing 3.34](#). Then run `bundle install` and verify that you can run Autotest using `bundle exec autotest`. *Extra credit:* Run `bundle install --binstubs` so that you can run Autotest using `bin/autotest`.

Listing 3.32. Code for a proposed Help page.

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

```
<h1>Help</h1>
<p>
  Get help on 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 3.33. The Pages controller spec with a base title.

`spec/controllers/pages_controller_spec.rb`

```
require 'spec_helper'

describe PagesController do
  render_views

  before(:each) do
    #
    # Define @base_title here.
```

```
#
end

describe "GET 'home'" do
  it "should be successful" do
    get 'home'
    response.should be_success
  end

  it "should have the right title" do
    get 'home'
    response.should have_selector("title",
                                  :content => @base_title + " | Home")
  end
end

describe "GET 'contact'" do
  it "should be successful" do
    get 'contact'
    response.should be_success
  end

  it "should have the right title" do
    get 'contact'
    response.should have_selector("title",
                                  :content => @base_title + " | Contact")
  end
end

describe "GET 'about'" do
  it "should be successful" do
    get 'about'
    response.should be_success
  end

  it "should have the right title" do
    get 'about'
    response.should have_selector("title",
                                  :content => @base_title + " | About")
  end
end
end
```

Listing 3.34. Adding Autotest to the Gemfile.

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

gem 'rails', '3.0.9'
gem 'sqlite3', '1.3.3'

group :development do
  gem 'rspec-rails', '2.6.1'
```



```

end

group :test do
  gem 'rspec-rails', '2.6.1'
  gem 'webrat', '0.7.1'
  # gem 'autotest', '4.4.6'
  # gem 'autotest-rails-pure', '4.1.2'
  # gem 'autotest-fsevent', '0.2.4'
  # gem 'autotest-growl', '0.2.9'
end

```

## « Chapter 2 A demo app Chapter 4 Rails-flavored Ruby »

1. As before, you may find the augmented file from [Listing 1.6](#) to be more convenient depending on your system. [↑](#)
2. 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.) [↑](#)
3. As usual, replace `mate` with the command for your text editor. [↑](#)
4. 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. [↑](#)
5. 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 Pages 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. [↑](#)
6. In the context of RSpec, TDD is also known as Behavior Driven Development, or BDD. (Frankly, I'm not convinced there's much of a difference.) [↑](#)
7. The [Shoulda](#) testing framework is a good alternate choice (and in fact can be used with RSpec). It's the Other Rails Way, so to speak. [↑](#)
8. This used to be just `autotest-rails`, but that gem depends on the full ZenTest suite, which caused problems on some systems. The `autotest-rails-pure` gem avoids this dependency. [↑](#)
9. These gems should properly be included in the `Gemfile` rather than installed at the command line, but in that case the `Gemfile` would be system-dependent—a situation I'd rather avoid in this tutorial. Incorporating Autotest into the `Gemfile` is left as an exercise ([Section 3.5](#)). [↑](#)
10. The Autotest Growl gem causes the test results to be automatically displayed to the monitor, whereas the FSEvent gem causes Autotest to use OS X filesystem events to trigger the test suite, rather than continuously polling the filesystem. Also note that with both gems you might need to use an updated version if you're running OS X Snow Leopard. [↑](#)
11. <http://fredschoeneman.posterous.com/pimp-your-autotest-notification> [↑](#)
12. In the context of RSpec, tests are often called *specs*, but for simplicity I'll usually stick to the term “test” —*except* when referring to a file such as `pages_controller_spec`, in which case I'll write “Pages controller spec”. [↑](#)
13. Most IDEs also have an interface to testing, but as noted in [Section 1.2.1](#) I have limited experience with those tools. [↑](#)
14. You can also run `bundle exec rake spec`, which is basically equivalent. (Annoyingly, if you want to run `rake spec` here you have to run `rake db:migrate` first, even though the tests in this chapter don't require a database.) [↑](#)
15. A *spork* is a combination spoon-fork. My guess is that the project's name is a pun on Spork's use of [POSIX forks](#). [↑](#)
16. DRb stands for “Distributed Ruby”. [↑](#)
17. We'll learn in [Section 4.3.3](#) that the `:content => "..."` syntax is a *hash* using a *symbol* as the key. [↑](#)

18. I consider this a step back from RSpec 1.3, which used `have_tag` in this context, which could be used to require an exact match. Unfortunately, as of this writing `have_tag` is not available in RSpec 2. [↑](#)
19. A newline is what comes at the end of a line, starting a, well, new line. In code, it is represented by the character `\n`. [↑](#)
20. Actually *counting* columns could drive you crazy, which is why many text editors have a visual aid to help you. Consider TextMate, 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`. [↑](#)
21. Rails 2.3/RSpec 1.3, used the shorter `have_tag` instead of `have_selector`, and the `:content` argument wasn't necessary either. Newer isn't always better... [↑](#)
22. In fact, the instance variable is actually visible in *any* view, a fact we'll make use of in [Section 8.2.2](#). [↑](#)
23. 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. If there is sufficient interest, I might produce a Rails Tutorial screencast series using Haml for the views. This would also allow for an introduction to [Sass](#), Haml's sister technology, which if anything is even more awesome than Haml. [↑](#)
24. 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, as far as developing web applications with Rails, it doesn't matter, and I've honestly never given the meaning of `<%= yield %>` a second thought—or even a first one. [↑](#)