Ruby on Rails Tutorial

Learn Rails by Example

Michael Hartl

Contents

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)
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, 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 program.

Copyright and license

```
 * ----------------------------------------------------------------------------
 * "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 seeming 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 sample_app
$ cd sample_app
```

As before, we'll copy the `.gitignore` file from Listing 1.2 to the sample app's Rails root directory:

```
$ git init
$ cp ~/rails_projects/first_app/.gitignore .
$ git add .
$ git commit -m "Initial commit"
```

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
```

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

```
$ heroku create
$ 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
```

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 1.3). 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 1.5).

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.[1] 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 3.1):[2]

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

Listing 3.1. A typical HTML file, with a friendly greeting.
`public/hello.html`

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html>
  <head>
    <title>Greeting</title>
  </head>
  <body>
    <p>Hello, world!</p>
  </body>
</html>
```

We see in Listing 3.1 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;3 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 1.5. 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.4 Rails actions come bundled together inside *controllers* (the C in MVC from Section 1.2.4), 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:[5]

Listing 3.2. Generating a Pages controller.

```
$ script/generate controller Pages home contact
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/pages
      exists  test/functional/
      create  test/unit/helpers/
      create  app/controllers/pages_controller.rb
      create  test/functional/pages_controller_test.rb
      create  app/helpers/pages_helper.rb
      create  test/unit/helpers/pages_helper_test.rb
      create  app/views/pages/home.html.erb
      create  app/views/pages/contact.html.erb
```

Here, I've intentionally "forgotten" the `about` page so that we can see how to add it in by hand (Section 3.2).

By default, Rails ensures that the actions inside controllers can be found at a URL of the following form:

```
http://localhost:3000/controller/action
```

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 `script/server`, and then navigate to `/pages/home` (Figure 3.4).

Figure 3.4: The raw home view (`/pages/home`) generated by Rails. (full size)

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 3.3. (You may note that, unlike the demo Users and Microposts controllers from Chapter 2, the Pages controller does not follow the REST conventions.)

Listing 3.3. The Pages controller made by Listing 3.2.
`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 1.2.4) 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 3.2, 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 3.4).

Listing 3.4. 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 3.5).

Listing 3.5. 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 -am "Added a Pages controller"
```

You may recall from Section 1.3.5 that we used the Git command `git commit -a -m "Message"`, with flags for "all changes" (`-a`) and a message (`-m`); Git also lets us roll the two flags into one as `-am`, and I'll stick with this more compact formulation throughout the rest of this book.

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.[6] 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, we'll be using a framework called RSpec, which is a *domain-specific language* for describing the behavior of code, together with a program (called `spec`) 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 Way*, has called RSpec "the Rails Way", and I agree.[7]

Installing RSpec

Since RSpec comes as a gem, installing it is easy:

```
$ [sudo] gem install rspec -v 1.3.0
$ [sudo] gem install rspec-rails -v 1.3.2
```

If you're using Ruby 1.8.7 or lower, RSpec should now be operational, but it won't work immediately if you're using Ruby 1.9. RSpec is built on top of the `Test::Unit` library, the default testing framework for Rails; up through Ruby 1.8.7, `Test::Unit` ships as part of the Ruby Standard Library, but (for reasons unknown to me) it is no longer included in Ruby 1.9. Moreover, RSpec doesn't yet work with the most recent standalone version of `Test::Unit`. This means that, if you are using Ruby 1.9, you should uninstall any existing versions of `Test::Unit` and then install the version RSpec needs:[8]

```
$ ruby -v
1.9.1
$ [sudo] gem uninstall test-unit
$ [sudo] gem install test-unit -v 1.2.3
```

To get RSpec started in a Rails application, we just need to run a script in the Rails root directory that generates some initial files:

```
$ script/generate rspec
```

Then add the files to the repository:

```
$ git add .
$ git status
# On branch static-pages
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   lib/tasks/rspec.rake
#       new file:   script/autospec
#       new file:   script/spec
#       new file:   spec/rcov.opts
#       new file:   spec/spec.opts
#       new file:   spec/spec_helper.rb
#
$ git commit -am "Added RSpec files"
```

Installing Autotest

Although not strictly necessary for testing, I've found that the Autotest tool is extraordinarily valuable for test-driven development. Autotest (part of the ZenTest suite) continuously runs your test suite in the background based on the specific file changes you make. 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), but for now you should install it if you want to try it:

```
$ [sudo] gem install autotest-rails -v 4.1.0
```

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:[9]

```
# On OS X
$ [sudo] gem install autotest-fsevent -v 0.1.1
$ [sudo] gem install autotest-growl -v 0.2.0
```

If FSEvent won't install properly, double-check that Xcode is installed on your system.

Then make an Autotest configuration file in your Rails root directory and fill it with the contents of Listing 3.6:

```
$ mate .autotest
```

(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".)

Listing 3.6. The `.autotest` configuration file for Autotest on OS X.

```
require "autotest/growl"
require "autotest/fsevent"
```

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 here.[10]

With that, we're ready to get testing!

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 make use of the `generate rspec_controller` script to get ourselves started. It works just like the `generate controller` script we saw in Section 3.1.2:

```
$ script/generate rspec_controller Pages home contact
      exists  app/controllers/
      exists  app/helpers/
      exists  app/views/pages
      create  spec/controllers/
      create  spec/helpers/
      create  spec/views/pages
      create  spec/controllers/pages_controller_spec.rb
      create  spec/helpers/pages_helper_spec.rb
   identical  app/controllers/pages_controller.rb
   identical  app/helpers/pages_helper.rb
      create  spec/views/pages/home.html.erb_spec.rb
   identical  app/views/pages/home.html.erb
      create  spec/views/pages/contact.html.erb_spec.rb
   identical  app/views/pages/contact.html.erb
```

Note that this step has created a directory called `spec` in our Rails root. This is to differentiate the RSpec tests from the ones that ship with Rails by default (using a framework called `Test::Unit`), which live in the `test` directory. Now that we are set up with RSpec, we won’t need that directory any longer, so it’s best to remove it:

```
$ rm -rf test/
```

In the output from `generate rspec_controller` above, you can see from lines like

```
   identical  app/controllers/pages_controller.rb
```

that this script does nothing to the files we created in Section 3.1.2, but it would have created them if they didn’t already exist.

By the way, I’m not partial to separate tests for views, which I’ve found to be quite brittle, so let’s remove them, together with the rarely used helper spec:

```
$ rm -rf spec/views
$ rm -f spec/helpers/pages_helper_spec.rb
```

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[11] we just generated (Listing 3.7).

Listing 3.7. 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
```

(Here I've deleted one of the generated examples and just left the ones we'll be using.)

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.

Let's focus on the first test 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` refers to one of the *HTTP verbs*, which are things that you can do with the hypertext transfer protocol (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 3.1).

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'll start learning about 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.

After the client sends a request, the 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.[12] One way to run all the tests is to use the `spec` script at the command line as follows:[13]

```
$ spec spec/
....

Finished in 0.183973 seconds

2 examples, 0 failures
```

(If any test fails, be sure that you've migrated the database with `rake db:migrate` as described in

Section 1.2.5.) In this command the first `spec` 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:

```
$ spec spec/controllers/
....
Finished in 0.18124 seconds

2 examples, 0 failures
```

You can also run a single file:

```
$ spec spec/controllers/pages_controller_spec.rb
....
Finished in 0.18328 seconds

2 examples, 0 failures
```

The results of all three commands 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 `spec 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 as follows:

```
$ autospec
```

If you're using a Mac with Growl notifications enabled, you should be able to replicate my setup, shown in Figure 3.5. With Autotest running in the background and Growl notifications telling you the status of your tests, TDD can be positively addictive.

Figure 3.5: Autotest (via `autospec`) in action, with a Growl notification. (full size)

Red

Now let's get to Red part of the Red-Green cycle by writing a failing test for the `about` page. Following the models from Listing 3.7, you can probably guess the right test (Listing 3.8).

Listing 3.8. The Pages controller spec with a failing test for the About page.
`spec/controllers/pages_controller_spec.rb`

```
require 'spec_helper'

describe PagesController do
  integrate_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 *integrate 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
  integrate_views
  .
  .
  .
end
```

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 3.6 (`spec spec/`) and Figure 3.7 (`autospec`). (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 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.)

Figure 3.6: Failing spec for the About page using `spec spec/`. (full size)

Figure 3.7: 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 3.3, let's first add an `about` action in the Pages controller (Listing 3.9).

Listing 3.9. 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 add the corresponding view. Eventually we'll fill it with something more informative, but for now we'll just mimic the content from the generated views (Listing 3.4 and Listing 3.5) for the `about` view (Listing 3.10).

Listing 3.10. 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 3.8) should get us back to Green:

```
$ spec spec/
```

Figure 3.8: 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 3.9).

Figure 3.9: 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 3.9 and Listing 3.10.)

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 3.1, 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	`/pages/home`	`"Ruby on Rails Tutorial Sample App"`	`" | Home"`
Contact	`/pages/contact`	`"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 3.8, adding title tests following the model in Listing 3.11.

Listing 3.11. A title test. 

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

This uses the `have_tag` method inside RSpec; the documentation for `have_tag` is surprisingly sparse, but what it does is not surprising at all: the code

```
response.should have_tag("title",
                         "Ruby on Rails Tutorial Sample App | Home")
```

checks to see that the material inside the `<title></title>` tags is `"Ruby on Rails Tutorial Sample App | Home"`. Note that I've broken the material inside `have_tag` into two lines; this tells you something important about Ruby syntax: Ruby doesn't care about newlines.[14] 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.[15]

Adding new tests for each of our three static pages following the model of Listing 3.11 gives us our new Pages controller spec (Listing 3.12).

Listing 3.12. The Pages controller spec with title tests.
`spec/controllers/pages_controller_spec.rb`

```
require 'spec_helper'

describe PagesController do
  integrate_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_tag("title",
                               "Ruby on Rails Tutorial Sample App | 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_tag("title",
                               "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_tag("title",
                               "Ruby on Rails Tutorial Sample App | About")
    end
  end
end
```

Note that the `integrate_views` line introduced in Listing 3.8 is necessary for the title tests to work.

With these tests in place, you should run

```
$ spec 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. (If you were very observant, you might have noticed the red X in the bottom corner of default Home page shown in Figure 3.9; this is the HTML Validator telling us we have a problem, which we'll fix in this section.) Let's start with the Home page (Listing 3.13), using the same basic HTML skeleton as in the "hello" page from Listing 3.1; we've just added one more element, a series of arguments inside the `html` tag giving more information about the language on the page (`en` for "English") and the XML namespace (`xmlns`).16

Listing 3.13. The view for the Home page with full HTML structure.
`app/views/pages/home.html.erb`

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html lang="en" xml:lang="en" xmlns="http://www.w3.org/1999/xhtml">
  <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://www.railstutorial.org/">Ruby on Rails Tutorial</a>
      sample application.
    </p>
  </body>
</html>
```

Listing 3.13 uses the title tested for in Listing 3.12:

```
<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 3.14 and Listing 3.15.

Listing 3.14. The view for the Contact page with full HTML structure.
`app/views/pages/contact.html.erb`

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html lang="en" xml:lang="en" xmlns="http://www.w3.org/1999/xhtml">
  <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://www.railstutorial.org/feedback">feedback page</a>.
    </p>
  </body>
</html>
```

Listing 3.15. The view for the About page with full HTML structure.
`app/views/pages/about.html.erb`

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html lang="en" xml:lang="en" xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Ruby on Rails Tutorial Sample App | About</title>
  </head>
  <body>
    <h1>About Us</h1>
    <p>
      <a href="http://www.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://www.railstutorial.org/">Ruby on Rails Tutorial</a>
```

You can see the results in Figure 3.10. Note the green checkmark at the lower right of the figure, indicating that the pages are now valid HTML.

Figure 3.10: 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.

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 3.16).

Listing 3.16. 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.17

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 3.17).

Listing 3.17. The view for the Home page with an Embedded Ruby title.
`app/views/pages/home.html.erb`

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
```

```
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html lang="en" xml:lang="en" xmlns="http://www.w3.org/1999/xhtml">
  <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://www.railstutorial.org/">Ruby on Rails Tutorial</a>
      sample application.
    </p>
  </body>
</html>
```

Listing 3.17 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.18 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 3.18 and Listing 3.19).

Listing 3.18. The view for the Contact page with an Embedded Ruby title.
`app/views/pages/contact.html.erb`

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html lang="en" xml:lang="en" xmlns="http://www.w3.org/1999/xhtml">
  <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://www.railstutorial.org/feedback">feedback page</a>.
    </p>
  </body>
</html>
```

Listing 3.19. The view for the About page with an Embedded Ruby title.
`app/views/pages/about.html.erb`

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html lang="en" xml:lang="en" xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Ruby on Rails Tutorial Sample App | <%= @title %></title>
  </head>
  <body>
    <h1>About Us</h1>
    <p>
      <a href="http://www.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 PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html lang="en" xml:lang="en" xmlns="http://www.w3.org/1999/xhtml">
  <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 `application.html.erb` and fill it with the contents of Listing 3.20.

Listing 3.20. The sample application site layout.
`app/views/layouts/application.html.erb`

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html lang="en" xml:lang="en" xmlns="http://www.w3.org/1999/xhtml">
```

```
  <head>
    <title>Ruby on Rails Tutorial Sample App | <%= @title %></title>
  </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;[19] 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 %>`.

Of course, our views 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 (Listing 3.21, Listing 3.22, and Listing 3.23).

Listing 3.21. 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://www.railstutorial.org/">Ruby on Rails Tutorial</a>
  sample application.
</p>
```

Listing 3.22. 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://www.railstutorial.org/feedback">feedback page</a>.
</p>
```

Listing 3.23. The About view with HTML structure removed.
`app/views/pages/about.html.erb`

```
<h1>About Us</h1>
<p>
  <a href="http://www.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 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 -am "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):

```
$ spec spec/
$ git push
```

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

```
$ spec spec/
$ git push heroku
```

Note that in both cases I've run `spec 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 3.24.
2. You may have noticed some repetition in the Pages controller spec (Listing 3.12). 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 3.25 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.)

Listing 3.24. 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://www.railstutorial.org/help">Rails Tutorial help page</a>.
  To get help on this sample app, see the
  <a href="http://www.railstutorial.org/book">Rails Tutorial book</a>.
</p>
```

Listing 3.25. The Pages controller spec with a base title.
`spec/controllers/pages_controller_spec.rb`

```
require 'spec_helper'

describe PagesController do
  integrate_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_tag("title", @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_tag("title", @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_tag("title", @base_title + " | About")
    end
  end
end
```

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

1. 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 Way* for more details.) ↑
2. As usual, replace `mate` with the command for your text editor. ↑
3. HTML changes with time; by explicitly making a doctype declaration we make it likelier that browsers will render our pages properly in the future. ↑
4. 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. ↑
5. Windows users will probably have to run `generate` explicitly with Ruby using `ruby script/generate`
   Throughout *Rails Tutorial*, I'll omit this explicit `ruby` for brevity. ↑
6. We'll be using RSpec for our tests, and in this context 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—the Other Rails Way, so to speak. ↑
8. Much thanks to reader Ron Green for going well beyond the call of duty to identify and solve this thorny problem. ↑
9. The most recent version of `autotest-fsevent`, 0.1.3, outputs verbose error messages on many versions of OS X, including mine. I recommend using version 0.1.1 until the problem is fixed. ↑
10. http://fredschoeneman.posterous.com/pimp-your-autotest-notification ↑
11. 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". ↑
12. Most IDEs also have an interface to testing, but as noted in Section 1.2.1 I have limited experience with those tools. ↑
13. You can also run `rake spec`, which is basically equivalent. When something goes wrong, `spec spec/` will show the stack trace, whereas `rake spec` doesn't show the stack trace by default. Since the stack trace is often useful in debugging, `spec spec/` is a prudent default. I confess, though, that out of habit I usually type `rake spec`, and switch to `spec spec/` if something goes wrong. ↑
14. 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`. ↑
15. 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`. ↑

16. Don’t worry about where these arguments come from or exactly what they mean; I don’t know either. I’m just in the habit of looking at the source of well-written web apps, and I’m not shy about borrowing their code. ↑
17. In fact, the instance variable is actually visible in *any* view, a fact we’ll make use of in Section 8.2.2. ↑
18. 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. ↑
19. 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. ↑