Covers Rails 3.2

Ruby on Rails Tutorial

Learn Web Development with Rails

Second Edition

Michael Hartl

Ruby on Rails Tutorial

Learn Web Development with Rails

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 the *Ruby on Rails Tutorial* is what I used to switch back to Rails again.

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

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

Enjoy!

Derek Sivers (sivers.org)
Formerly: Founder, CD Baby
Currently: Founder, Thoughts Ltd.

Acknowledgments

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

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

About the author

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

Copyright and license

Chapter 1

From zero to deploy

Welcome to the *Ruby on Rails Tutorial*. The goal of this book is to be the best answer to the question, "If I want to learn web development with Ruby on Rails, where should I start?" By the time you finish the *Ruby on Rails Tutorial*, you will have all the skills you need to develop and deploy your own custom web applications with Rails. You will also be ready to benefit from the many more advanced books, blogs, and screencasts that are part of the thriving Rails educational ecosystem. Finally, since the *Ruby on Rails Tutorial* uses Rails 3, the knowledge you gain here represents the state of the art in web development. (The most up-to-date version of the *Ruby on Rails Tutorial* can be found on the book's website at http://railstutorial.org/; if you are reading this book offline, be sure to check the online version of the Rails Tutorial book at http://railstutorial.org/book for the latest updates.)

Note that the goal of this book is *not* merely to teach Rails, but rather to teach *web development with Rails*, which means acquiring (or expanding) the skills needed to develop software for the World Wide Web. In addition to Ruby on Rails, this skillset includes HTML & CSS, databases, version control, testing, and deployment. To accomplish this goal, the *Ruby on Rails Tutorial* takes an integrated approach: you will learn Rails by example by building a substantial sample application from scratch. As Derek Sivers notes in the foreword, this book is structured as a linear narrative, designed to be read from start to finish. If you are used to skipping around in technical books, taking this linear approach might require some adjustment, but I suggest giving it a try. You can

think of the *Ruby on Rails Tutorial* as a video game where you are the main character, and where you level up as a Rails developer in each chapter. (The exercises are the minibosses.)

In this first chapter, we'll get started with Ruby on Rails by installing all the necessary software and by setting up our development environment (Section 1.2). We'll then create our first Rails application, called (appropriately enough) `first_app`. The *Rails Tutorial* emphasizes good software development practices, so immediately after creating our fresh new Rails project we'll put it under version control with Git (Section 1.3). And, believe it or not, in this chapter we'll even put our first app on the wider web by *deploying* it to production (Section 1.4).

In Chapter 2, we'll make a second project, whose purpose is to demonstrate the basic workings of a Rails application. To get up and running quickly, we'll build this *demo app* (called `demo_app`) using scaffolding (Box 1.1) to generate code; since this code is both ugly and complex, Chapter 2 will focus on interacting with the demo app through its *URIs* (sometimes called *URLs*)[1] using a web browser.

The rest of the tutorial focuses on developing a single large *sample application* (called `sample_app`), writing all the code from scratch. We'll develop the sample app using *test-driven development* (TDD), getting started in Chapter 3 by creating static pages and then adding a little dynamic content. We'll take a quick detour in Chapter 4 to learn a little about the Ruby language underlying Rails. Then, in Chapter 5 through Chapter 9, we'll complete the foundation for the sample application by making a site layout, a user data model, and a full registration and authentication system. Finally, in Chapter 10 and Chapter 11 we'll add microblogging and social features to make a working example site.

The final sample application will bear more than a passing resemblance to a certain popular social microblogging site—a site which, coincidentally, was also originally written in Rails. Though of necessity our efforts will focus on this specific sample application, the emphasis throughout the *Rails Tutorial* will be on general principles, so that you will have a solid foundation no matter

[1] *URI* stands for Uniform Resource Identifier, while the slightly less general *URL* stands for Uniform Resource Locator. In practice, the URI is usually equivalent to "the thing you see in the address bar of your browser".

what kinds of web applications you want to build.

Box 1.1. Scaffolding: Quicker, easier, more seductive

From the beginning, Rails has benefited from a palpable sense of excitement, starting with the famous 15-minute weblog video by Rails creator David Heinemeier Hansson. That video and its successors are a great way to get a taste of Rails' power, and I recommend watching them. But be warned: they accomplish their amazing fifteen-minute feat using a feature called *scaffolding*, which relies heavily on *generated code*, magically created by the Rails `generate` command.

When writing a Ruby on Rails tutorial, it is tempting to rely on the scaffolding approach—it's quicker, easier, more seductive. But the complexity and sheer amount of code in the scaffolding can be utterly overwhelming to a beginning Rails developer; you may be able to use it, but you probably won't understand it. Following the scaffolding approach risks turning you into a virtuoso script generator with little (and brittle) actual knowledge of Rails.

In the *Ruby on Rails Tutorial*, we'll take the (nearly) polar opposite approach: although Chapter 2 will develop a small demo app using scaffolding, the core of the *Rails Tutorial* is the sample app, which we'll start writing in Chapter 3. At each stage of developing the sample application, we will write *small, bite-sized* pieces of code—simple enough to understand, yet novel enough to be challenging. The cumulative effect will be a deeper, more flexible knowledge of Rails, giving you a good background for writing nearly any type of web application.

1.1 Introduction

Since its debut in 2004, Ruby on Rails has rapidly become one of the most powerful and popular frameworks for building dynamic web applications. Everyone from scrappy startups to huge companies have used Rails: 37signals, GitHub, Shopify, Scribd, Twitter, LivingSocial, Groupon, Hulu, the Yellow

Pages—the list of sites using Rails goes on and on. There are also many web development shops that specialize in Rails, such as ENTP, thoughtbot, Pivotal Labs, and Hashrocket, plus innumerable independent consultants, trainers, and contractors.

What makes Rails so great? First of all, Ruby on Rails is 100% open-source, available under the permissive MIT License, and as a result it also costs nothing to download or use. Rails also owes much of its success to its elegant and compact design; by exploiting the malleability of the underlying Ruby language, Rails effectively creates a domain-specific language for writing web applications. As a result, many common web programming tasks—such as generating HTML, making data models, and routing URIs—are easy with Rails, and the resulting application code is concise and readable.

Rails also adapts rapidly to new developments in web technology and framework design. For example, Rails was one of the first frameworks to fully digest and implement the REST architectural style for structuring web applications (which we'll be learning about throughout this tutorial). And when other frameworks develop successful new techniques, Rails creator David Heinemeier Hansson and the Rails core team don't hesitate to incorporate their ideas. Perhaps the most dramatic example is the merger of Rails and Merb, a rival Ruby web framework, so that Rails now benefits from Merb's modular design, stable API, and improved performance.

Finally, Rails benefits from an unusually enthusiastic and diverse community. The results include hundreds of open-source contributors, well-attended conferences, a huge number of plugins and gems (self-contained solutions to specific problems such as pagination and image upload), a rich variety of informative blogs, and a cornucopia of discussion forums and IRC channels. The large number of Rails programmers also makes it easier to handle the inevitable application errors: the "Google the error message" algorithm nearly always produces a relevant blog post or discussion-forum thread.

1.1.1 Comments for various readers

The *Rails Tutorial* contains integrated tutorials not only for Rails, but also for the underlying Ruby language, the RSpec testing framework, HTML, CSS, a

small amount of JavaScript, and even a little SQL. This means that, no matter where you currently are in your knowledge of web development, by the time you finish this tutorial you will be ready for more advanced Rails resources, as well as for the more systematic treatments of the other subjects mentioned. It also means that there's a *lot* of material to cover; if you don't already have much experience programming computers, you might find it overwhelming. The comments below contain some suggestions for approaching the *Rails Tutorial* depending on your background.

All readers: One common question when learning Rails is whether to learn Ruby first. The answer depends on your personal learning style and how much programming experience you already have. If you prefer to learn everything systematically from the ground up, or if you have never programmed before, then learning Ruby first might work well for you, and in this case I recommend *Beginning Ruby* by Peter Cooper. On the other hand, many beginning Rails developers are excited about making *web* applications, and would rather not slog through a 500-page book on pure Ruby before ever writing a single web page. In this case, I recommend following the short interactive tutorial at Try Ruby,[2] and then optionally do the free tutorial at Rails for Zombies[3] to get a taste of what Rails can do.

Another common question is whether to use tests from the start. As noted in the introduction, the *Rails Tutorial* uses test-driven development (also called test-first development), which in my view is the best way to develop Rails applications, but it does introduce a substantial amount of overhead and complexity. If you find yourself getting bogged down by the tests, I suggest either skipping them on a first reading or (even better) using them as a tool to verify your code's correctness without worrying about how they work. This latter strategy involves creating the necessary test files (called *specs*) and filling them with the test code *exactly* as it appears in the book. You can then run the test suite (as described in Chapter 5) to watch it fail, then write the application code as described in the tutorial, and finally re-run the test suite to watch it pass.

[2] http://tryruby.org/

[3] http://railsforzombies.org/

Inexperienced programmers: The *Rails Tutorial* is not aimed principally at beginning programmers, and web applications, even relatively simple ones, are by their nature fairly complex. If you are completely new to web programming and find the *Rails Tutorial* too difficult, I suggest learning the basics of HTML and CSS and then giving the *Rails Tutorial* another go. (Unfortunately, I don't have a personal recommendation here, but *Head First HTML* looks promising, and one reader recommends *CSS: The Missing Manual* by David Sawyer McFarland.) You might also consider reading the first few chapters of *Beginning Ruby* by Peter Cooper, which starts with sample applications much smaller than a full-blown web app. That said, a surprising number of beginners have used this tutorial to learn web development, so I suggest giving it a try, and I especially recommend the *Rails Tutorial* screencast series[4] to give you an "over-the-shoulder" look at Rails software development.

Experienced programmers new to web development: Your previous experience means you probably already understand ideas like classes, methods, data structures, etc., which is a big advantage. Be warned that if your background is in C/C++ or Java, you may find Ruby a bit of an odd duck, and it might take time to get used to it; just stick with it and eventually you'll be fine. (Ruby even lets you put semicolons at the ends of lines if you miss them too much.) The *Rails Tutorial* covers all the web-specific ideas you'll need, so don't worry if you don't currently know a `PUT` from a `POST`.

Experienced web developers new to Rails: You have a great head start, especially if you have used a dynamic language such as PHP or (even better) Python. The basics of what we cover will likely be familiar, but test-driven development may be new to you, as may be the structured REST style favored by Rails. Ruby has its own idiosyncrasies, so those will likely be new, too.

Experienced Ruby programmers: The set of Ruby programmers who don't know Rails is a small one nowadays, but if you are a member of this elite group you can fly through this book and then move on to *The Rails 3 Way* by Obie

[4] http://railstutorial.org/screencasts

Fernandez.

Inexperienced Rails programmers: You've perhaps read some other tutorials and made a few small Rails apps yourself. Based on reader feedback, I'm confident that you can still get a lot out of this book. Among other things, the techniques here may be more up-to-date than the ones you picked up when you originally learned Rails.

Experienced Rails programmers: This book is unnecessary for you, but many experienced Rails developers have expressed surprise at how much they learned from this book, and you might enjoy seeing Rails from a different perspective.

After finishing the *Ruby on Rails Tutorial*, I recommend that experienced programmers read *The Well-Grounded Rubyist* by David A. Black, which is an excellent in-depth discussion of Ruby from the ground up, or *The Ruby Way* by Hal Fulton, which is also fairly advanced but takes a more topical approach. Then move on to *The Rails 3 Way* to deepen your Rails expertise.

At the end of this process, no matter where you started, you should be ready for the many more intermediate-to-advanced Rails resources out there. Here are some I particularly recommend:

- RailsCasts by Ryan Bates: Excellent (mostly) free Rails screencasts
- PeepCode: Excellent commercial screencasts
- Code School: Interactive programming courses
- Rails Guides: Good topical and up-to-date Rails references
- RailsCasts by Ryan Bates: Did I already mention RailsCasts? Seriously: *RailsCasts*.

1.1.2 "Scaling" Rails

Before moving on with the rest of the introduction, I'd like to take a moment to address the one issue that dogged the Rails framework the most in its early days: the supposed inability of Rails to "scale"—i.e., to handle large amounts of traffic. Part of this issue relied on a misconception; you scale a *site*, not a framework, and Rails, as awesome as it is, is only a framework. So the real question should have been, "Can a site built with Rails scale?" In any case, the question has now been definitively answered in the affirmative: some of the most heavily trafficked sites in the world use Rails. Actually *doing* the scaling is beyond the scope of just Rails, but rest assured that if *your* application ever needs to handle the load of Hulu or the Yellow Pages, Rails won't stop you from taking over the world.

1.1.3 Conventions in this book

The conventions in this book are mostly self-explanatory. In this section, I'll mention some that may not be.

Both the HTML and PDF editions of this book are full of links, both to internal sections (such as Section 1.2) and to external sites (such as the main Ruby on Rails download page).[5]

Many examples in this book use command-line commands. For simplicity, all command line examples use a Unix-style command line prompt (a dollar sign), as follows:

```
$ echo "hello, world"
hello, world
```

Windows users should understand that their systems will use the analogous angle prompt `>`:

[5] When reading the *Rails Tutorial*, you may find it convenient to follow an internal section link to look at the reference and then immediately go back to where you were before. This is easy when reading the book as a web page, since you can just use the Back button of your browser, but both Adobe Reader and OS X's Preview allow you to do this with the PDF as well. In Reader, you can right-click on the document and select "Previous View" to go back. In Preview, use the Go menu: `Go > Back`.

```
C:\Sites> echo "hello, world"
hello, world
```

On Unix systems, some commands should be executed with **sudo**, which stands for "substitute user do".[6] By default, a command executed with **sudo** is run as an administrative user, which has access to files and directories that normal users can't touch, such as in this example from Section 1.2.2:

```
$ sudo ruby setup.rb
```

Most Unix/Linux/OS X systems require **sudo** by default, unless you are using Ruby Version Manager as suggested in Section 1.2.2; in this case, you would type this instead:

```
$ ruby setup.rb
```

Rails comes with lots of commands that can be run at the command line. For example, in Section 1.2.5 we'll run a local development web server as follows:

```
$ rails server
```

As with the command-line prompt, the *Rails Tutorial* uses the Unix convention for directory separators (i.e., a forward slash **/**). My Rails Tutorial sample application, for instance, lives in

[6] Many people erroneously believe that **sudo** stands for "superuser do" because it runs commands as the superuser (root) by default. In fact, **sudo** is a concatenation of the **su** command and the English word "do", and **su** stands for "substitute user", as you can verify by typing **man su** in your shell.

```
/Users/mhartl/rails_projects/sample_app
```

On Windows, the analogous directory would be

```
C:\Sites\sample_app
```

The root directory for any given app is known as the *Rails root*, but this terminology is confusing and many people mistakenly believe that the "Rails root" is the root directory for Rails itself. For clarity, the *Rails Tutorial* will refer to the Rails root as the *application root*, and henceforth all directories will be relative to this directory. For example, the **config** directory of my sample application is

```
/Users/mhartl/rails_projects/sample_app/config
```

The application root directory here is everything before **config**, i.e.,

```
/Users/mhartl/rails_projects/sample_app
```

For brevity, when referring to the file

```
/Users/mhartl/rails_projects/sample_app/config/routes.rb
```

I'll omit the application root and simply write **config/routes.rb**.

The *Rails Tutorial* often shows output from various programs (shell commands, version control status, Ruby programs, etc.). Because of the innumerable small differences between different computer systems, the output you see may not always agree exactly with what is shown in the text, but this is not cause for concern.

Some commands may produce errors depending on your system; rather than attempt the Sisyphean task of documenting all such errors in this tutorial, I will delegate to the "Google the error message" algorithm, which among other things is good practice for real-life software development. If you run into any problems while following the tutorial, I suggest consulting the resources listed on the Rails Tutorial help page.[7]

1.2 Up and running

> I think of Chapter 1 as the "weeding out phase" in law school—if you can get your dev environment set up, the rest is easy to get through.
> —Bob Cavezza, *Rails Tutorial* reader

It's time now to get going with a Ruby on Rails development environment and our first application. There is quite a bit of overhead here, especially if you don't have extensive programming experience, so don't get discouraged if it takes a while to get started. It's not just you; every developer goes through it (often more than once), but rest assured that the effort will be richly rewarded.

1.2.1 Development environments

Considering various idiosyncratic customizations, there are probably as many development environments as there are Rails programmers, but there are at least two broad types: text editor/command line environments, and integrated development environments (IDEs). Let's consider the latter first.

IDEs

There is no shortage of Rails IDEs, including RadRails, RubyMine, and 3rd Rail. I've heard especially good things about RubyMine, and one reader (David

[7] http://railstutorial.org/help

Loeffler) has assembled notes on how to use RubyMine with this tutorial.[8] If you're comfortable using an IDE, I suggest taking a look at the options mentioned to see what fits with the way you work.

Text editors and command lines

Instead of using an IDE, I prefer to use a *text editor* to edit text, and a *command line* to issue commands (Figure 1.1). Which combination you use depends on your tastes and your platform.

- **Text editor:** I recommend Sublime Text 2, an outstanding cross-platform text editor that is in beta as of this writing but has already proven to be exceptionally powerful. Sublime Text is heavily influenced by TextMate, and in fact is compatible with most TextMate customizations, such as snippets and color schemes. (TextMate, which is available only on OS X, is still a good choice if you use a Mac.) A second excellent choice is Vim,[9] versions of which are available for all major platforms. Sublime Text can be obtained commercially, whereas Vim can be obtained at no cost; both are industrial-strength editors, but in my experience Sublime Text is *much* more accessible to beginners.

- **Terminal:** On OS X, I recommend either use iTerm or the native Terminal app. On Linux, the default terminal is fine. On Windows, many users prefer to develop Rails applications in a virtual machine running Linux, in which case your command-line options reduce to the previous case. If developing within Windows itself, I recommend using the command prompt that comes with Rails Installer (Section 1.2.2).

If you decide to use Sublime Text, you want to follow the setup instructions for Rails Tutorial Sublime Text.[10] *Note*: Such configuration settings are fiddly and error-prone, so this step should only be attempted by advanced users.

[8] https://github.com/perfectionist/sample_project/wiki

[9] The vi editor is one of the most ancient yet powerful weapons in the Unix arsenal, and Vim is "vi improved".

[10] https://github.com/mhartl/rails_tutorial_sublime_text

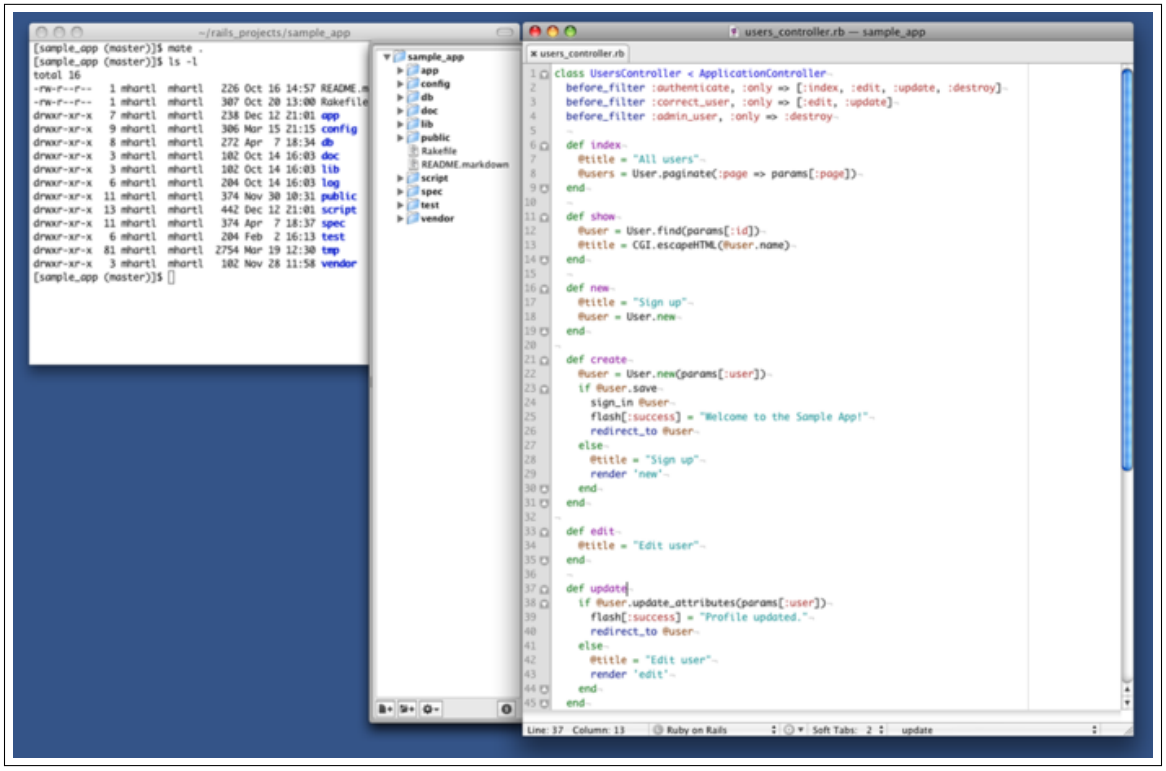

Figure 1.1: A text editor/command line development environment (TextMate/iTerm). (full size)

Browsers

Although there are many web browsers to choose from, the vast majority of Rails programmers use Firefox, Safari, or Chrome when developing. The screenshots in Rails Tutorial will generally be of a Firefox browser. If you use Firefox, I suggest using the Firebug add-on, which lets you perform all sorts of magic, such as dynamically inspecting (and even editing) the HTML structure and CSS rules on any page. For those not using Firefox, both Safari and Chrome have a built-in "Inspect element" feature available by right-clicking on any part of the page.

A note about tools

In the process of getting your development environment up and running, you may find that you spend a *lot* of time getting everything just right. The learning process for editors and IDEs is particularly long; you can spend weeks on Sublime Text or Vim tutorials alone. If you're new to this game, I want to assure you that *spending time learning tools is normal*. Everyone goes through it. Sometimes it is frustrating, and it's easy to get impatient when you have an awesome web app in your head and you *just want to learn Rails already*, but have to spend a week learning some weird ancient Unix editor just to get started. But a craftsman has to know his tools, and in the end the reward is worth the effort.

1.2.2 Ruby, RubyGems, Rails, and Git

> Practically all the software in the world is either broken or very difficult to use. So users dread software. They've been trained that whenever they try to install something, or even fill out a form online, it's not going to work. *I* dread installing stuff, and I have a Ph.D. in computer science.
> —Paul Graham, *Founders at Work*

Now it's time to install Ruby and Rails. I've done my best to cover as many bases as possible, but systems vary, and many things can go wrong during these

steps. Be sure to Google the error message or consult the Rails Tutorial help page if you run into trouble.

Unless otherwise noted, you should use the exact versions of all software used in the tutorial, including Rails itself, if you want the same results. Sometimes minor version differences will yield identical results, but you shouldn't count on this, especially with respect to Rails versions. The main exception is Ruby itself: 1.9.2 and 1.9.3 are virtually identical for the purposes of this tutorial, so feel free to use either one.

Rails Installer (Windows)

Installing Rails on Windows used to be a real pain, but thanks to the efforts of the good people at Engine Yard—especially Dr. Nic Williams and Wayne E. Seguin—installing Rails and related software on Windows is now easy. If you are using Windows, go to Rails Installer and download the Rails Installer executable and view the excellent installation video. Double-click the executable and follow the instructions to install Git (so you can skip Section 1.2.2), Ruby (skip Section 1.2.2), RubyGems (skip Section 1.2.2), and Rails itself (skip Section 1.2.2). Once the installation has finished, you can skip right to the creation of the first application in Section 1.2.3.

Bear in mind that the Rails Installer might use a slightly different version of Rails from the one installed in Section 1.2.2, which might cause incompatibilities. To fix this, I am currently working with Nic and Wayne to create a list of Rails Installers ordered by Rails version number.

Install Git

Much of the Rails ecosystem depends in one way or another on a version control system called Git (covered in more detail in Section 1.3). Because its use is ubiquitous, you should install Git even at this early stage; I suggest following the installation instructions for your platform at the Installing Git section of *Pro Git*.

Install Ruby

The next step is to install Ruby. It's possible that your system already has it; try running

```
$ ruby -v
ruby 1.9.3
```

to see the version number. Rails 3 requires Ruby 1.8.7 or later and works best with Ruby 1.9.x. This tutorial assumes that most readers are using Ruby 1.9.2 or 1.9.3, but Ruby 1.8.7 should work as well (although there is one syntax difference, covered in Chapter 4, and assorted minor differences in output).

As part of installing Ruby, if you are using OS X or Linux I strongly recommend using Ruby Version Manager (RVM), which allows you to install and manage multiple versions of Ruby on the same machine. (The Pik project accomplishes a similar feat on Windows.) This is particularly important if you want to run different versions of Ruby or Rails on the same machine. If you run into any problems with RVM, you can often find its creator, Wayne E. Seguin, on the RVM IRC channel (#rvm on freenode.net).[11] If you are running Linux, I particularly recommend How to install Ruby on Rails in Ubuntu on the Sudobits Blog.

After installing RVM, you can install Ruby as follows:[12]

```
$ rvm get head && rvm reload
$ rvm install 1.9.3
<wait a while>
```

Here the first command updates and reloads RVM itself, which is a good practice since RVM gets updated frequently. The second installs the 1.9.3 version

[11] If you haven't used IRC before, I suggest you start by searching the web for "irc client <your platform>". Two good native clients for OS X are Colloquy and LimeChat. And of course there's always the web interface at http://webchat.freenode.net/?channels=rvm.

[12] You might have to install the Subversion version control system to get this to work.

of Ruby; depending on your system, they might take a while to download and compile, so don't worry if it seems to be taking forever.

Some OS X users have trouble with the lack of an `autoconf` executable, which you can fix by installing Homebrew[13] (a package management system for OS X) and then running

```
$ brew install automake
$ rvm install 1.9.3
```

Some Linux users report having to include the path to a library called OpenSSL:

```
$ rvm install 1.9.3 --with-openssl-dir=$HOME/.rvm/
```

On some older OS X systems, you might have to include the path to the readline library:

```
$ rvm install 1.9.3 --with-readline-dir=/opt/local
```

(Like I said, lots of things can go wrong. The only solution is web searches and determination.)

After installing Ruby, you should configure your system for the other software needed to run Rails applications. This typically involves installing *gems*, which are self-contained packages of Ruby code. Since gems with different version numbers sometimes conflict, it is often convenient to create separate *gemsets*, which are self-contained bundles of gems. For the purposes of this tutorial, I suggest creating a gemset called `rails3tutorial2ndEd`:

[13] http://mxcl.github.com/homebrew/

```
$ rvm use 1.9.3@rails3tutorial2ndEd --create --default
Using /Users/mhartl/.rvm/gems/ruby-1.9.3 with gemset rails3tutorial2ndEd
```

This command creates (`--create`) the gemset **`rails3tutorial2ndEd`** associated with Ruby 1.9.3 while arranging to start using it immediately (`use`) and setting it as the default (`--default`) gemset, so that any time we open a new terminal window the **`1.9.3@rails3tutorial2ndEd`** Ruby/gemset combination is automatically selected. RVM supports a large variety of commands for manipulating gemsets; see the documentation at http://rvm.beginrescueend.com/gemsets/. If you ever get stuck with RVM, running commands like these should help you get your bearings:

```
$ rvm --help
$ rvm gemset --help
```

Install RubyGems

RubyGems is a package manager for Ruby projects, and there are many useful libraries (including Rails) available as Ruby packages, or *gems*. Installing RubyGems should be easy once you install Ruby. In fact, if you have installed RVM, you already have RubyGems, since RVM includes it automatically:

```
$ which gem
/Users/mhartl/.rvm/rubies/ruby-1.9.3-p0/bin/gem
```

If you don't already have it, you should download RubyGems, extract it, and then go to the **`rubygems`** directory and run the setup program:

```
$ ruby setup.rb
```

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

If you already have RubyGems installed, you should make sure your system uses the version used in this tutorial:

```
$ gem update --system 1.8.24
```

Freezing your system to this particular version will help prevent conflicts as RubyGems changes in the future.

When installing gems, by default RubyGems generates two different kinds of documentation (called ri and rdoc), but many Ruby and Rails developers find that the time to build them isn't worth the benefit. (Many programmers rely on online documentation instead of the native ri and rdoc documents.) To prevent the automatic generation of the documentation, I recommend making a gem configuration file called **.gemrc** in your home directory as in Listing 1.1 with the line in Listing 1.2. (The tilde "~" means "home directory", while the dot **.** in **.gemrc** makes the file hidden, which is a common convention for configuration files.)

Listing 1.1. Creating a gem configuration file.

```
$ subl ~/.gemrc
```

Here **subl** is the command-line command to launch Sublime Text on OS X, which you can set up using the Sublime Text 2 documentation for the OS X command line. If you're on a different platform, or if you're using a different editor, you should replace this command as necessary (i.e., by double-clicking the application icon or by using an alternate command such as **mate**, **vim**, **gvim**, or **mvim**). For brevity, throughout the rest of this tutorial I'll use **subl** as a shorthand for "open with your favorite text editor."

Listing 1.2. Suppressing the ri and rdoc documentation in **.gemrc**.

```
install: --no-rdoc --no-ri
update: --no-rdoc --no-ri
```

Install Rails

Once you've installed RubyGems, installing Rails should be easy. This tutorial standardizes on Rails 3.2, which we can install as follows:

```
$ gem install rails -v 3.2.11
```

To check your Rails installation, run the following command to print out the version number:

```
$ rails -v
Rails 3.2.11
```

Note: If you installed Rails using the Rails Installer in Section 1.2.2, there might be slight version differences. As of this writing, those differences are not relevant, but in the future, as the current Rails version diverges from the one used in this tutorial, these differences may become significant. I am currently working with Engine Yard to create links to specific versions of the Rails Installer.

If you're running Linux, you might have to install a couple of other packages at this point:

```
$ sudo apt-get install libxslt-dev libxml2-dev libsqlite3-dev # Linux only
```

1.2.3 The first application

Virtually all Rails applications start the same way, with the **`rails`** command. This handy program creates a skeleton Rails application in a directory of your

choice. To get started, make a directory for your Rails projects and then run the **rails** command to make the first application (Listing 1.3):

Listing 1.3. Running **rails** to generate a new application.

```
$ mkdir rails_projects
$ cd rails_projects
$ rails new first_app
      create
      create  README.rdoc
      create  Rakefile
      create  config.ru
      create  .gitignore
      create  Gemfile
      create  app
      create  app/assets/images/rails.png
      create  app/assets/javascripts/application.js
      create  app/assets/stylesheets/application.css
      create  app/controllers/application_controller.rb
      create  app/helpers/application_helper.rb
      create  app/mailers
      create  app/models
      create  app/views/layouts/application.html.erb
      create  app/mailers/.gitkeep
      create  app/models/.gitkeep
      create  config
      create  config/routes.rb
      create  config/application.rb
      create  config/environment.rb
      .
      .
      .
      create  vendor/plugins
      create  vendor/plugins/.gitkeep
         run  bundle install
Fetching source index for https://rubygems.org/
.
.
.
Your bundle is complete! Use `bundle show [gemname]` to see where a bundled
gem is installed.
```

As seen at the end of Listing 1.3, running **rails** automatically runs the **bundle install** command after the file creation is done. If that step doesn't work right now, don't worry; follow the steps in Section 1.2.4 and you should be able to get it to work.

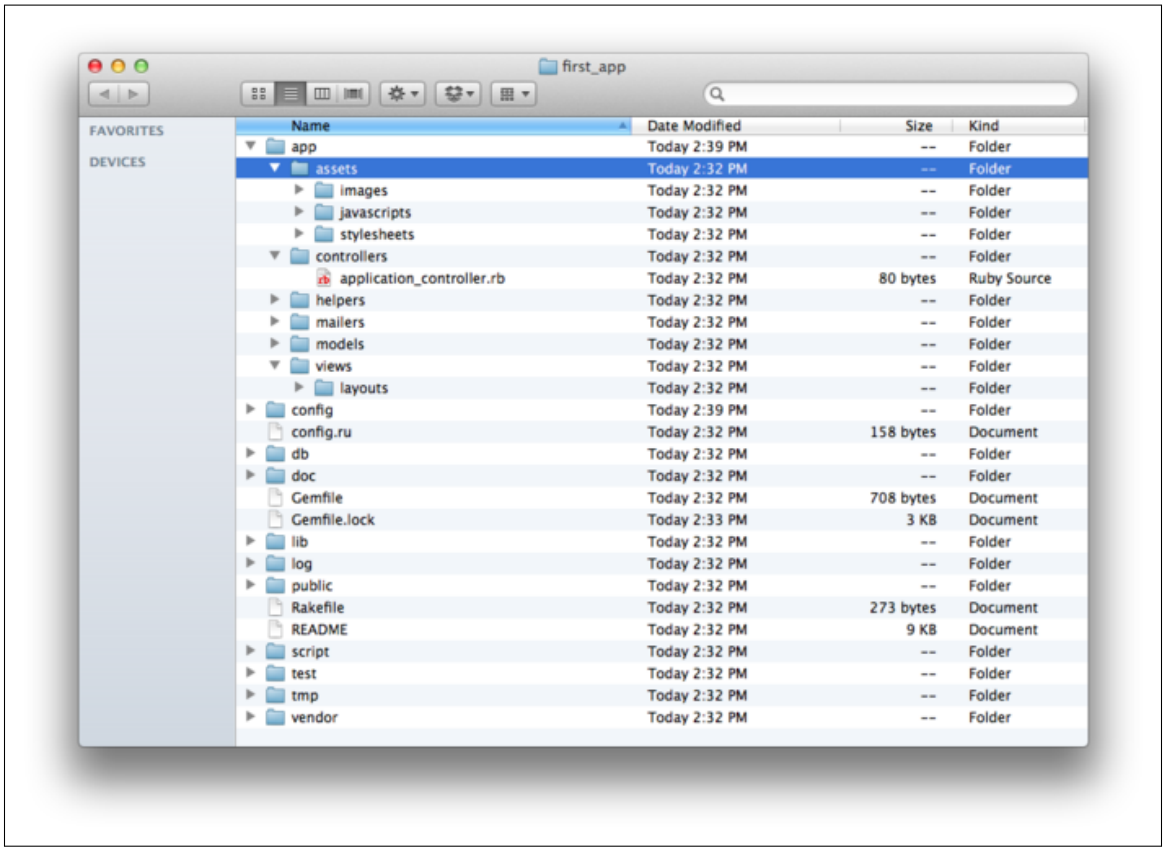

Figure 1.2: The directory structure for a newly hatched Rails app. (full size)

Notice how many files and directories the `rails` command creates. This standard directory and file structure (Figure 1.2) is one of the many advantages of Rails; it immediately gets you from zero to a functional (if minimal) application. Moreover, since the structure is common to all Rails apps, you can immediately get your bearings when looking at someone else's code. A summary of the default Rails files appears in Table 1.1; we'll learn about most of these files and directories throughout the rest of this book. In particular, starting in Section 5.2.1 we'll discuss the `app/assets` directory, part of the *asset pipeline* (new as of Rails 3.1) that makes it easier than ever to organize and deploy assets such as cascading style sheets and JavaScript files.

File/Directory	Purpose
`app/`	Core application (app) code, including models, views, controllers, and helpers
`app/assets`	Applications assets such as cascading style sheets (CSS), JavaScript files, and images
`config/`	Application configuration
`db/`	Database files
`doc/`	Documentation for the application
`lib/`	Library modules
`lib/assets`	Library assets such as cascading style sheets (CSS), JavaScript files, and images
`log/`	Application log files
`public/`	Data accessible to the public (e.g., web browsers), such as error pages
`script/rails`	A script for generating code, opening console sessions, or starting a local server
`test/`	Application tests (made obsolete by the `spec/` directory in Section 3.1.2)
`tmp/`	Temporary files
`vendor/`	Third-party code such as plugins and gems
`vendor/assets`	Third-party assets such as cascading style sheets (CSS), JavaScript files, and images
`README.rdoc`	A brief description of the application
`Rakefile`	Utility tasks available via the `rake` command
`Gemfile`	Gem requirements for this app
`Gemfile.lock`	A list of gems used to ensure that all copies of the app use the same gem versions
`config.ru`	A configuration file for Rack middleware
`.gitignore`	Patterns for files that should be ignored by Git

Table 1.1: A summary of the default Rails directory structure.

1.2.4 Bundler

After creating a new Rails application, the next step is to use *Bundler* to install and include the gems needed by the app. As noted briefly in Section 1.2.3, Bundler is run automatically (via **`bundle install`**) by the **`rails`** command, but in this section we'll make some changes to the default application gems and run Bundler again. This involves opening the **`Gemfile`** with your favorite text editor:

```
$ cd first_app/
$ subl Gemfile
```

The result should look something like Listing 1.4. The code in this file is Ruby, but don't worry at this point about the syntax; Chapter 4 will cover Ruby in more depth.

Listing 1.4. The default **`Gemfile`** in the **`first_app`** directory.

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

gem 'rails', '3.2.11'

# Bundle edge Rails instead:
# gem 'rails', :git => 'git://github.com/rails/rails.git'

gem 'sqlite3'


# Gems used only for assets and not required
# in production environments by default.
group :assets do
  gem 'sass-rails',   '~> 3.2.3'
  gem 'coffee-rails', '~> 3.2.2'

  gem 'uglifier', '>= 1.2.3'
end

gem 'jquery-rails'

# To use ActiveModel has_secure_password
# gem 'bcrypt-ruby', '~> 3.0.0'
```

```
# To use Jbuilder templates for JSON
# gem 'jbuilder'

# Use unicorn as the web server
# gem 'unicorn'

# Deploy with Capistrano
# gem 'capistrano'

# To use debugger
# gem 'ruby-debug19', :require => 'ruby-debug'
```

Many of these lines are commented out with the hash symbol **#**; they are there to show you some commonly needed gems and to give examples of the Bundler syntax. For now, we won't need any gems other than the defaults: Rails itself, some gems related to the asset pipeline (Section 5.2.1), the gem for the jQuery JavaScript library, and the gem for the Ruby interface to the SQLite database.

Unless you specify a version number to the **gem** command, Bundler will automatically install the latest version of the gem. Unfortunately, gem updates often cause minor but potentially confusing breakage, so in this tutorial we'll include explicit version numbers known to work, as seen in Listing 1.5 (which also omits the commented-out lines from Listing 1.4).

Listing 1.5. A **Gemfile** with an explicit version of each Ruby gem.

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

gem 'rails', '3.2.11'

group :development do
  gem 'sqlite3', '1.3.5'
end


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

  gem 'uglifier', '1.2.3'
end
```

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

Listing 1.5 changes the line for jQuery, the default JavaScript library used by Rails, from

```
gem 'jquery-rails'
```

to

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

We've also changed

```
gem 'sqlite3'
```

to

```
group :development do
  gem 'sqlite3', '1.3.5'
end
```

which forces Bundler to install version `1.3.5` of the `sqlite3` gem. Note that we've also taken this opportunity to arrange for SQLite to be included only in a development environment (Section 7.1.1), which prevents potential conflicts with the database used by Heroku (Section 1.4).

Listing 1.5 also changes a few other lines, converting

```
group :assets do
  gem 'sass-rails',   '~> 3.2.3'
  gem 'coffee-rails', '~> 3.2.2'
  gem 'uglifier', '>= 1.2.3'
end
```

to

```
group :assets do
  gem 'sass-rails',   '3.2.5'
  gem 'coffee-rails', '3.2.2'
  gem 'uglifier', '1.2.3'
end
```

The syntax

```
gem 'uglifier', '>= 1.2.3'
```

installs the latest version of the **uglifier** gem (which handles file compression for the asset pipeline) as long as it's greater than version **1.2.3**—even if it's, say, version **7.2**. Meanwhile, the code

```
gem 'coffee-rails', '~> 3.2.2'
```

installs the gem **coffee-rails** (also needed by the asset pipeline) as long as it's lower than version **3.3**. In other words, the >= notation always performs upgrades, whereas the ~> 3.2.2 notation only performs upgrades to minor point releases (e.g., from **3.1.1** to **3.1.2**), but not to major point releases (e.g., from **3.1** to **3.2**). Unfortunately, experience shows that even minor point releases often break things, so for the *Rails Tutorial* we'll err on the side of caution by including exact version numbers for virtually all gems. (The only exception is gems that are in release candidate or beta stage as of this writing; for those gems, we'll use ~> so that the final versions will be loaded once they're done.)

Once you've assembled the proper **Gemfile**, install the gems using **bundle install**:

```
$ bundle install
Fetching source index for https://rubygems.org/
.
.
.
```

If you're running OS X and you get an error about missing Ruby header files (e.g., `ruby.h`) at this point, you may need to install Xcode. These are developer tools that came with your OS X installation disk, but to avoid the full installation I recommend the much smaller Command Line Tools for Xcode.[14] If you get a libxslt error when installing the Nokogiri gem, try reinstalling Ruby:

```
$ rvm reinstall 1.9.3
$ bundle install
```

The `bundle install` command might take a few moments, but when it's done our application will be ready to run. *Note:* This setup is fine for the first app, but it isn't ideal. Chapter 3 covers a more powerful (and slightly more advanced) method for installing Ruby gems with Bundler.

1.2.5 `rails server`

Thanks to running `rails new` in Section 1.2.3 and `bundle install` in Section 1.2.4, we already have an application we can run—but how? Happily, Rails comes with a command-line program, or *script*, that runs a *local* web server, visible only from your development machine:[15]

```
$ rails server
=> Booting WEBrick
=> Rails application starting on http://0.0.0.0:3000
=> Call with -d to detach
=> Ctrl-C to shutdown server
```

[14] https://developer.apple.com/downloads/

[15] Recall from Section 1.1.3 that Windows users might have to type `ruby rails server` instead.

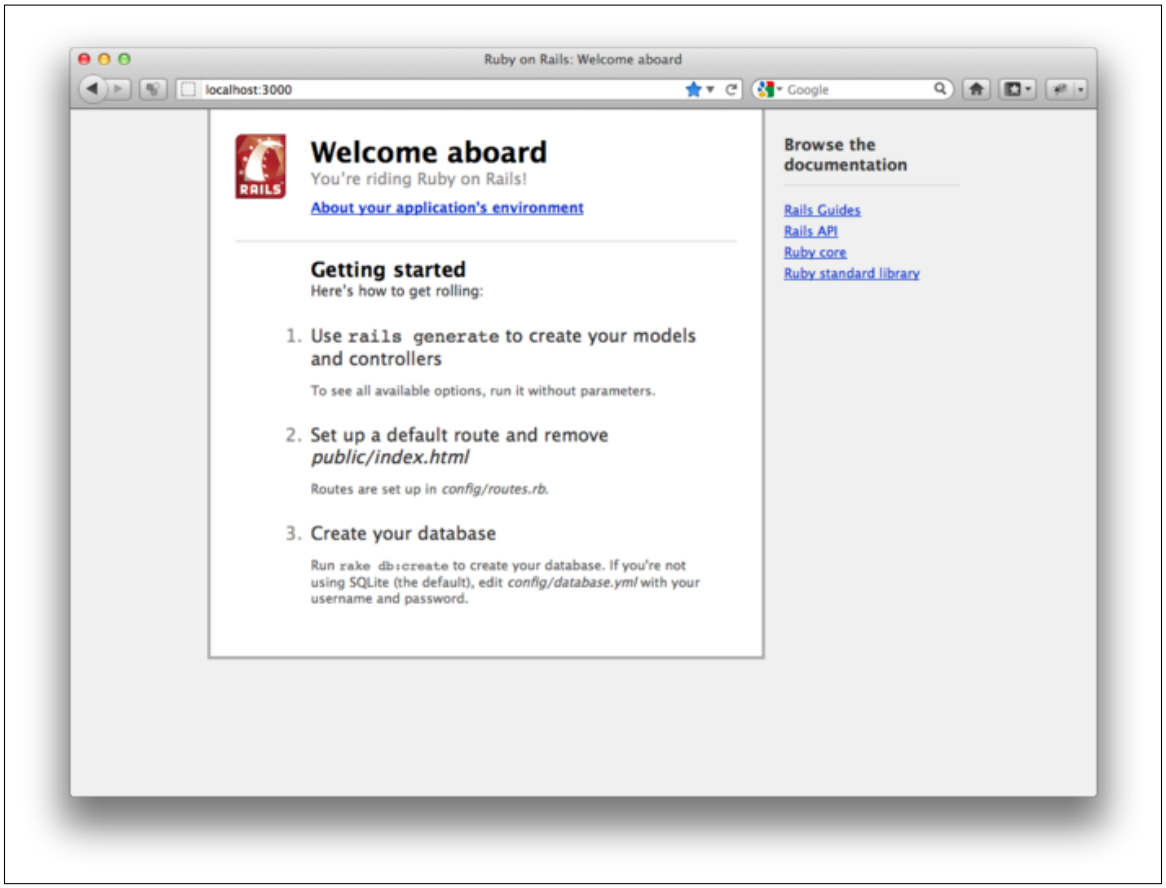

Figure 1.3: The default Rails page. (full size)

(If your system complains about the lack of a JavaScript runtime, visit the execjs page at GitHub for a list of possibilities. I particularly recommend installing Node.js.) This tells us that the application is running on port number 3000[16] at the address `0.0.0.0`. This address tells the computer to listen on every available IP address configured on that specific machine; in particular, we can view the application using the special address `127.0.0.1`, which is also known as `localhost`. We can see the result of visiting http://localhost:3000/ in Figure 1.3.

[16]Normally, websites run on port 80, but this usually requires special privileges, so Rails picks a less restricted higher-numbered port for the development server.

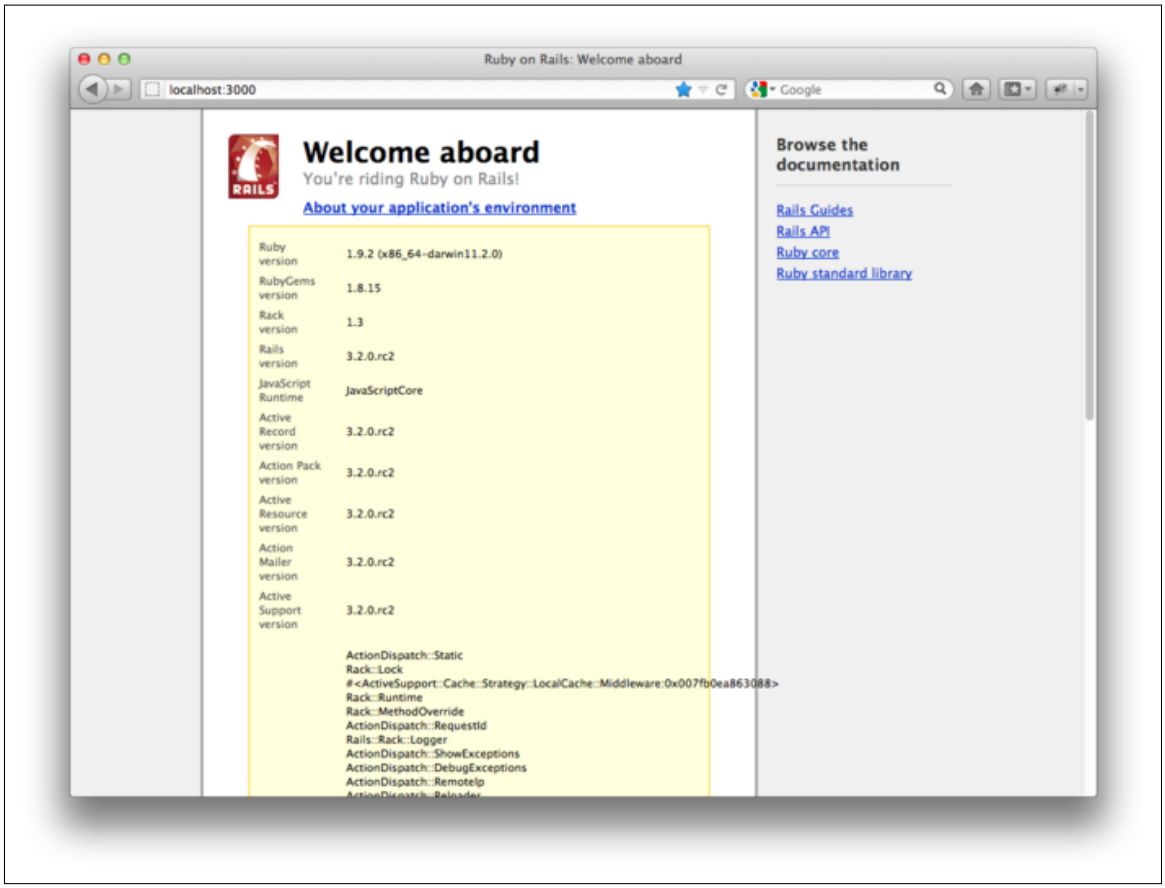

Figure 1.4: The default page with the app environment. (full size)

To see information about our first application, click on the link "About your application's environment". The result is shown in Figure 1.4. (Figure 1.4 represents the environment on my machine when I made the screenshot; your results may differ.)

Of course, we don't need the default Rails page in the long run, but it's nice to see it working for now. We'll remove the default page (and replace it with a custom home page) in Section 5.3.2.

1.2.6 Model-view-controller (MVC)

Even at this early stage, it's helpful to get a high-level overview of how Rails applications work (Figure 1.5). You might have noticed that the standard Rails application structure (Figure 1.2) has an application directory called `app/` with three subdirectories: `models`, `views`, and `controllers`. This is a hint that Rails follows the model-view-controller (MVC) architectural pattern, which enforces a separation between "domain logic" (also called "business logic") from the input and presentation logic associated with a graphical user interface (GUI). In the case of web applications, the "domain logic" typically consists of data models for things like users, articles, and products, and the GUI is just a web page in a web browser.

When interacting with a Rails application, a browser sends a *request*, which is received by a web server and passed on to a Rails *controller*, which is in charge of what to do next. In some cases, the controller will immediately render a *view*, which is a template that gets converted to HTML and sent back to the browser. More commonly for dynamic sites, the controller interacts with a *model*, which is a Ruby object that represents an element of the site (such as a user) and is in charge of communicating with the database. After invoking the model, the controller then renders the view and returns the complete web page to the browser as HTML.

If this discussion seems a bit abstract right now, worry not; we'll refer back to this section frequently. In addition, Section 2.2.2 has a more detailed discussion of MVC in the context of the demo app. Finally, the sample app will use all aspects of MVC; we'll cover controllers and views starting in Section 3.1.2, models starting in Section 6.1, and we'll see all three working together in Section 7.1.2.

1.3 Version control with Git

Now that we have a fresh and working Rails application, we'll take a moment for a step that, while technically optional, would be viewed by many Rails developers as practically essential, namely, placing our application source code under *version control*. Version control systems allow us to track changes to our

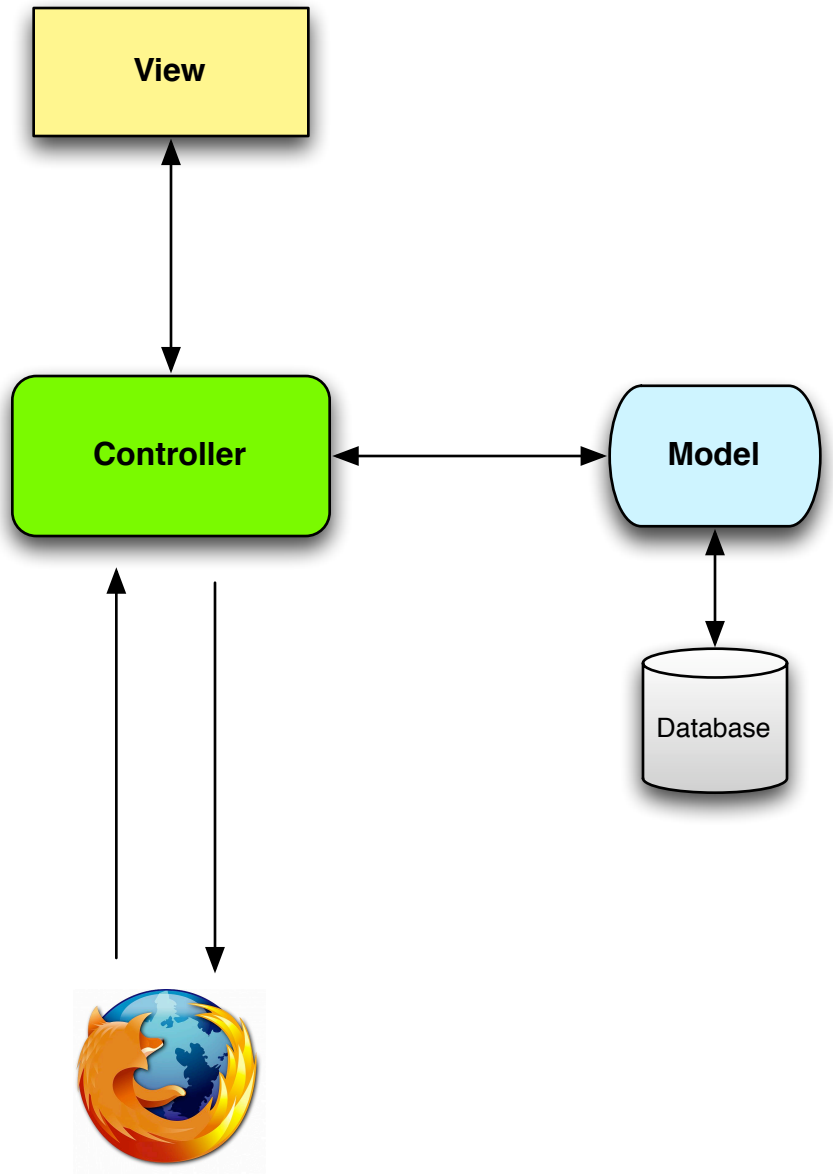

Figure 1.5: A schematic representation of the model-view-controller (MVC) architecture.

project's code, collaborate more easily, and roll back any inadvertent errors (such as accidentally deleting files). Knowing how to use a version control system is a required skill for every software developer.

There are many options for version control, but the Rails community has largely standardized on Git, a distributed version control system originally developed by Linus Torvalds to host the Linux kernel. Git is a large subject, and we'll only be scratching the surface in this book, but there are many good free resources online; I especially recommend *Pro Git* by Scott Chacon (Apress, 2009). Putting your source code under version control with Git is *strongly* recommended, not only because it's nearly a universal practice in the Rails world, but also because it will allow you to share your code more easily (Section 1.3.4) and deploy your application right here in the first chapter (Section 1.4).

1.3.1 Installation and setup

The first step is to install Git if you haven't yet followed the steps in Section 1.2.2. (As noted in that section, this involves following the instructions in the Installing Git section of *Pro Git*.)

First-time system setup

After installing Git, you should perform a set of one-time setup steps. These are *system* setups, meaning you only have to do them once per computer:

```
$ git config --global user.name "Your Name"
$ git config --global user.email your.email@example.com
```

I also like to use `co` in place of the more verbose `checkout` command, which we can arrange as follows:

```
$ git config --global alias.co checkout
```

This tutorial will usually use the full `checkout` command, which works for systems that don't have `co` configured, but in real life I nearly always use `git co`.

As a final setup step, you can optionally set the editor Git will use for commit messages. If you use a graphical editor such as Sublime Text, TextMate, gVim, or MacVim, you need to use a flag to make sure that the editor stays attached to the shell instead of detaching immediately:[17]

```
$ git config --global core.editor "subl -w"
```

Replace `"subl -w"` with `"mate -w"` for TextMate, `"gvim -f"` for gVim, or `"mvim -f"` for MacVim.

First-time repository setup

Now we come to some steps that are necessary each time you create a new *repository*. First navigate to the root directory of the first app and initialize a new repository:

```
$ git init
Initialized empty Git repository in /Users/mhartl/rails_projects/first_app/.git/
```

The next step is to add the project files to the repository. There's a minor complication, though: by default Git tracks the changes of *all* the files, but there are some files we don't want to track. For example, Rails creates log files to record the behavior of the application; these files change frequently, and we don't want our version control system to have to update them constantly. Git has a simple mechanism to ignore such files: simply include a file

[17] Normally this is a feature, since it lets you continue to use the command line after launching your editor, but Git interprets the detachment as closing the file with an empty commit message, which prevents the commit from going through. I only mention this point because it can be seriously confusing if you try to set your editor to `subl` or `gvim` without the flag. If you find this note confusing, feel free to ignore it.

called `.gitignore` in the application root directory with some rules telling Git which files to ignore.[18]

Looking again at Table 1.1, we see that the `rails` command creates a default `.gitignore` file in the application root directory, as shown in Listing 1.6.

Listing 1.6. The default `.gitignore` created by the `rails` command.

```
# See http://help.github.com/ignore-files/ for more about ignoring files.
#
# If you find yourself ignoring temporary files generated by your text editor
# or operating system, you probably want to add a global ignore instead:
#   git config --global core.excludesfile ~/.gitignore_global

# Ignore bundler config
/.bundle

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

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

Listing 1.6 causes Git to ignore files such as log files, Rails temporary (`tmp`) files, and SQLite databases. (For example, to ignore log files, which live in the `log/` directory, we use `log/*.log` to ignore all files that end in `.log`.) Most of these ignored files change frequently and automatically, so including them under version control is inconvenient; moreover, when collaborating with others they can cause frustrating and irrelevant conflicts.

The `.gitignore` file in Listing 1.6 is probably sufficient for this tutorial, but depending on your system you may find Listing 1.7 more convenient. This augmented `.gitignore` arranges to ignore Rails documentation files, Vim and Emacs swap files, and (for OS X users) the weird `.DS_Store` directories created by the Mac Finder application. If you want to use this broader set of ignored files, open up `.gitignore` in your favorite text editor and fill it with the contents of Listing 1.7.

[18] If you can't see the `.gitignore` file in your directory, you may need to configure your directory viewer to show hidden files.

Listing 1.7. An augmented `.gitignore` file.

```
# Ignore bundler config
/.bundle

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

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

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

1.3.2 Adding and committing

Finally, we'll add the files in your new Rails project to Git and then commit the results. You can add all the files (apart from those that match the ignore patterns in `.gitignore`) as follows:

```
$ git add .
```

Here the dot '`.`' represents the current directory, and Git is smart enough to add the files *recursively*, so it automatically includes all the subdirectories. This command adds the project files to a *staging area*, which contains pending changes to your project; you can see which files are in the staging area using the `status` command:[19]

[19] If in the future any unwanted files start showing up when you type `git status`, just add them to your `.gitignore` file from Listing 1.7.

```
$ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   README.rdoc
#       new file:   Rakefile
.
.
.
```

(The results are long, so I've used vertical dots to indicate omitted output.)

To tell Git you want to keep the changes, use the **commit** command:

```
$ git commit -m "Initial commit"
[master (root-commit) df0a62f] Initial commit
42 files changed, 8461 insertions(+), 0 deletions(-)
create mode 100644 README.rdoc
create mode 100644 Rakefile
.
.
.
```

The **-m** flag lets you add a message for the commit; if you omit **-m**, Git will open the editor you set in Section 1.3.1 and have you enter the message there.

It is important to note that Git commits are *local*, recorded only on the machine on which the commits occur. This is in contrast to the popular open-source version control system called Subversion, in which a commit necessarily makes changes on a remote repository. Git divides a Subversion-style commit into its two logical pieces: a local recording of the changes (**git commit**) and a push of the changes up to a remote repository (**git push**). We'll see an example of the push step in Section 1.3.5.

By the way, you can see a list of your commit messages using the **log** command:

```
$ git log
commit df0a62f3f091e53ffa799309b3e32c27b0b38eb4
Author: Michael Hartl <michael@michaelhartl.com>
Date:   Thu Oct 15 11:36:21 2009 -0700

  Initial commit
```

To exit `git log`, you may have to type `q` to quit.

1.3.3 What good does Git do you?

It's probably not entirely clear at this point why putting your source under version control does you any good, so let me give just one example. (We'll see many others in the chapters ahead.) Suppose you've made some accidental changes, such as (D'oh!) deleting the critical `app/controllers/` directory:

```
$ ls app/controllers/
application_controller.rb
$ rm -rf app/controllers/
$ ls app/controllers/
ls: app/controllers/: No such file or directory
```

Here we're using the Unix `ls` command to list the contents of the `app/controllers/` directory and the `rm` command to remove it. The `-rf` flag means "recursive force", which recursively removes all files, directories, subdirectories, and so on, without asking for explicit confirmation of each deletion.

Let's check the status to see what's up:

```
$ git status
# On branch master
# Changed but not updated:
#   (use "git add/rm <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       deleted:    app/controllers/application_controller.rb
#
no changes added to commit (use "git add" and/or "git commit -a")
```

We see here that a file has been deleted, but the changes are only on the "working tree"; they haven't been committed yet. This means we can still undo the changes easily by having Git check out the previous commit with the `checkout` command (and a `-f` flag to force overwriting the current changes):

```
$ git checkout -f
$ git status
# On branch master
nothing to commit (working directory clean)
$ ls app/controllers/
application_controller.rb
```

The missing directory and file are back. That's a relief!

1.3.4 GitHub

Now that you've put your project under version control with Git, it's time to push your code up to GitHub, a social code site optimized for hosting and sharing Git repositories. Putting a copy of your Git repository at GitHub serves two purposes: it's a full backup of your code (including the full history of commits), and it makes any future collaboration much easier. This step is optional, but being a GitHub member will open the door to participating in a wide variety of open-source projects.

GitHub has a variety of paid plans, but for open-source code their services are free, so sign up for a free GitHub account if you don't have one already. (You might have to follow the GitHub tutorial on creating SSH keys first.) After signing up, click on the link to create a repository and fill in the information as in Figure 1.6. (Take care *not* to initialize the repository with a `README` file, as `rails new` creates one of those automatically.) After submitting the form, push up your first application as follows:

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

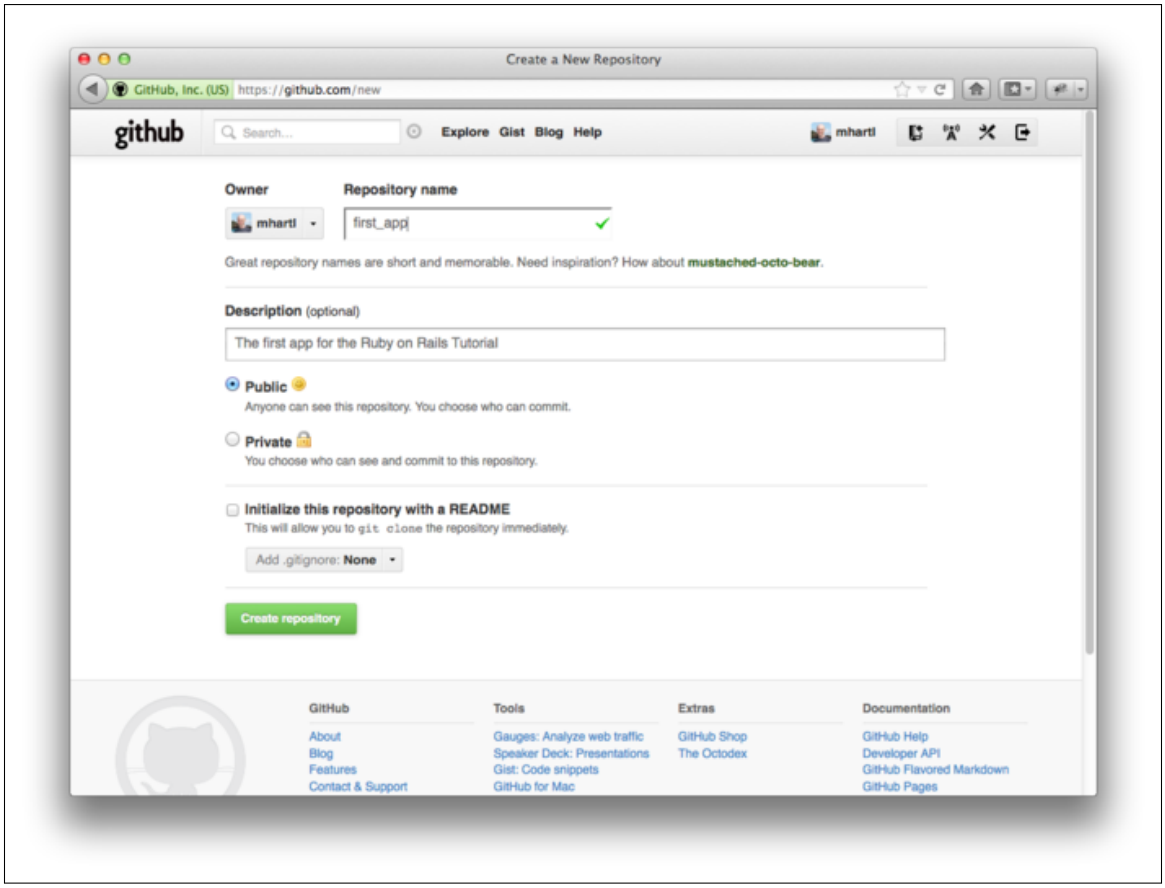

Figure 1.6: Creating the first app repository at GitHub. (full size)

These commands tell Git that you want to add GitHub as the origin for your main (*master*) branch and then push your repository up to GitHub. (Don't worry about what the `-u` flag does; if you're curious, do a web search for "git set upstream".) Of course, you should replace `<username>` with your actual username. For example, the command I ran for the **railstutorial** user was

```
$ git remote add origin git@github.com:railstutorial/first_app.git
```

The result is a page at GitHub for the first application repository, with file browsing, full commit history, and lots of other goodies (Figure 1.7).

GitHub also has native applications to augment the command-line interface, so if you're more comfortable with GUI apps you might want to check out GitHub for Windows or GitHub for Mac. (GitHub for Linux is still just Git, it seems.)

1.3.5 Branch, edit, commit, merge

If you've followed the steps in Section 1.3.4, you might notice that GitHub automatically shows the contents of the **README** file on the main repository page. In our case, since the project is a Rails application generated using the **rails** command, the **README** file is the one that comes with Rails (Figure 1.8). Because of the **.rdoc** extension on the file, GitHub ensures that it is formatted nicely, but the contents aren't helpful at all, so in this section we'll make our first edit by changing the **README** to describe our project rather than the Rails framework itself. In the process, we'll see a first example of the branch, edit, commit, merge workflow that I recommend using with Git.

Branch

Git is incredibly good at making *branches*, which are effectively copies of a repository where we can make (possibly experimental) changes without modifying the parent files. In most cases, the parent repository is the *master* branch, and we can create a new topic branch by using **checkout** with the **-b** flag:

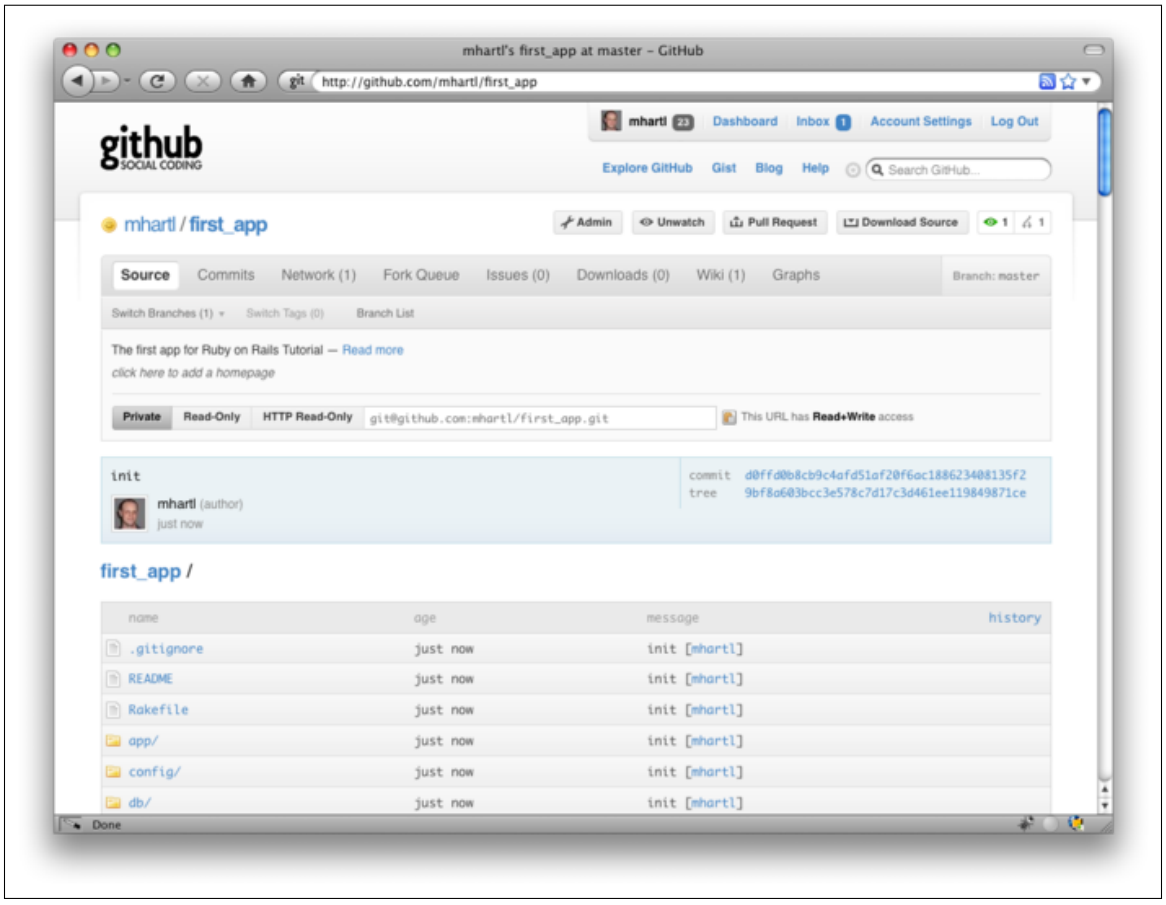

Figure 1.7: A GitHub repository page. (full size)

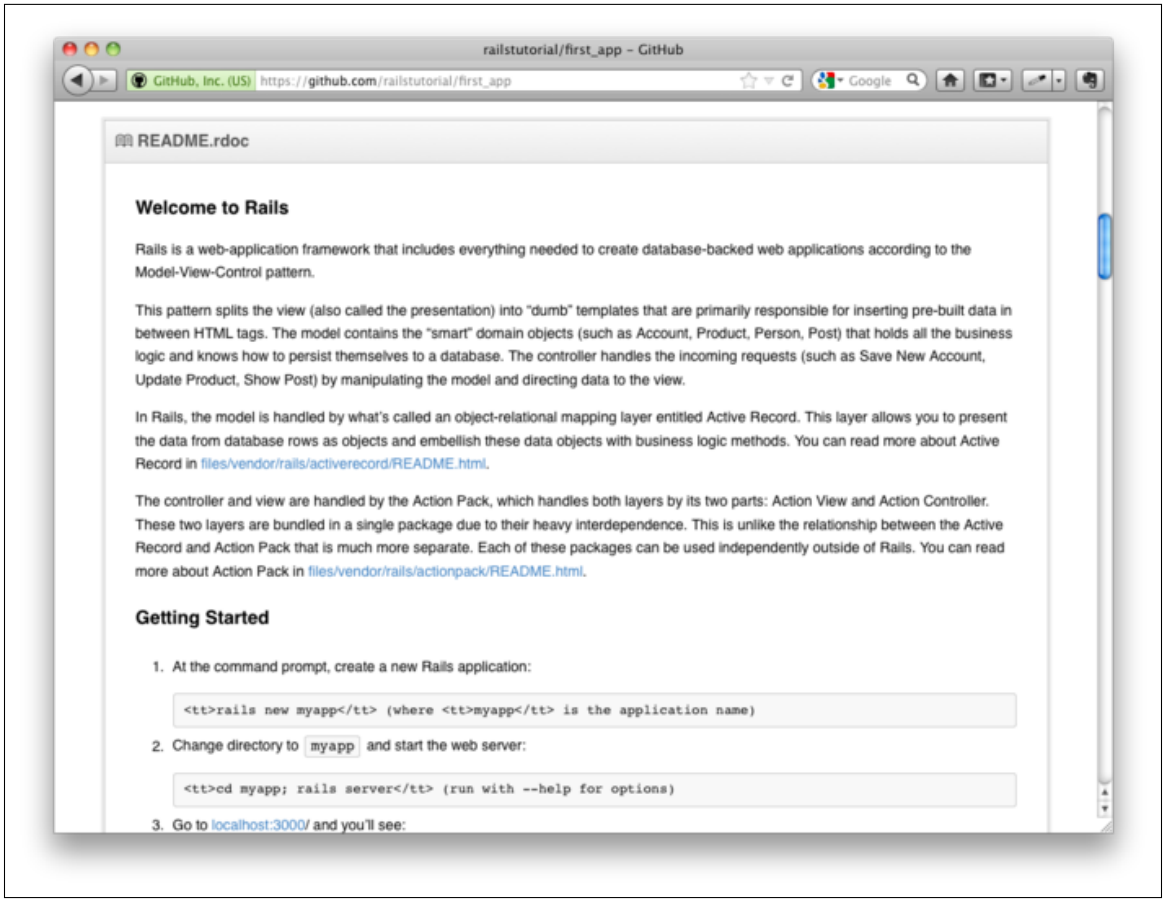

Figure 1.8: The initial (rather useless) **README** file for our project at GitHub. (full size)

```
$ git checkout -b modify-README
Switched to a new branch 'modify-README'
$ git branch
master
* modify-README
```

Here the second command, **`git branch`**, just lists all the local branches, and the asterisk **`*`** identifies which branch we're currently on. Note that **`git checkout -b modify-README`** both creates a new branch and switches to it, as indicated by the asterisk in front of the **`modify-README`** branch. (If you set up the **`co`** alias in Section 1.3, you can use **`git co -b modify-README`** instead.)

The full value of branching only becomes clear when working on a project with multiple developers,[20] but branches are helpful even for a single-developer tutorial such as this one. In particular, the master branch is insulated from any changes we make to the topic branch, so even if we *really* screw things up we can always abandon the changes by checking out the master branch and deleting the topic branch. We'll see how to do this at the end of the section.

By the way, for a change as small as this one I wouldn't normally bother with a new branch, but it's never too early to start practicing good habits.

Edit

After creating the topic branch, we'll edit it to make it a little more descriptive. I prefer the Markdown markup language to the default RDoc for this purpose, and if you use the file extension **`.md`** then GitHub will automatically format it nicely for you. So, first we'll use Git's version of the Unix **`mv`** ("move") command to change the name, and then fill it in with the contents of Listing 1.8:

```
$ git mv README.rdoc README.md
$ subl README.md
```

[20] See the chapter Git Branching in *Pro Git* for details.

Listing 1.8. The new `README` file, `README.md`.

```
# Ruby on Rails Tutorial: first application

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

Commit

With the changes made, we can take a look at the status of our branch:

```
$ git status
# On branch modify-README
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       renamed:    README.rdoc -> README.md
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   README.md
#
```

At this point, we could use `git add .` as in Section 1.3.2, but Git provides the `-a` flag as a shortcut for the (very common) case of committing all modifications to existing files (or files created using `git mv`, which don't count as new files to Git):

```
$ git commit -a -m "Improve the README file"
2 files changed, 5 insertions(+), 243 deletions(-)
delete mode 100644 README.rdoc
create mode 100644 README.md
```

Be careful about using the `-a` flag improperly; if you have added any new files to the project since the last commit, you still have to tell Git about them using `git add` first.

Note that we write the commit message in the *present* tense. Git models commits as a series of patches, and in this context it makes sense to describe what each commit *does*, rather than what it did. Moreover, this usage matches up with the commit messages generated by Git commands themselves. See the GitHub post Shiny new commit styles for more information.

Merge

Now that we've finished making our changes, we're ready to *merge* the results back into our master branch:

```
$ git checkout master
Switched to branch 'master'
$ git merge modify-README
Updating 34f06b7..2c92bef
Fast forward
README.rdoc     |  243 ----------------------------------------------------
README.md       |    5 +
2 files changed, 5 insertions(+), 243 deletions(-)
delete mode 100644 README.rdoc
create mode 100644 README.md
```

Note that the Git output frequently includes things like **`34f06b7`**, which are related to Git's internal representation of repositories. Your exact results will differ in these details, but otherwise should essentially match the output shown above.

After you've merged in the changes, you can tidy up your branches by deleting the topic branch using **`git branch -d`** if you're done with it:

```
$ git branch -d modify-README
Deleted branch modify-README (was 2c92bef).
```

This step is optional, and in fact it's quite common to leave the topic branch intact. This way you can switch back and forth between the topic and master branches, merging in changes every time you reach a natural stopping point.

As mentioned above, it's also possible to abandon your topic branch changes, in this case with **`git branch -D`**:

Figure 1.9: The improved **README** file formatted with Markdown. (full size)

```
# For illustration only; don't do this unless you mess up a branch
$ git checkout -b topic-branch
$ <really screw up the branch>
$ git add .
$ git commit -a -m "Major screw up"
$ git checkout master
$ git branch -D topic-branch
```

Unlike the **-d** flag, the **-D** flag will delete the branch even though we haven't merged in the changes.

Push

Now that we've updated the **README**, we can push the changes up to GitHub to see the result. Since we have already done one push (Section 1.3.4), on most systems we can omit **origin master**, and simply run **git push**:

```
$ git push
```

As promised, GitHub nicely formats the new file using Markdown (Figure 1.9).

1.4 Deploying

Even at this early stage, we're already going to deploy our (still-empty) Rails application to production. This step is optional, but deploying early and of-

ten allows us to catch any deployment problems early in our development cycle. The alternative—deploying only after laborious effort sealed away in a development environment—often leads to terrible integration headaches when launch time comes.[21]

Deploying Rails applications used to be a pain, but the Rails deployment ecosystem has matured rapidly in the past few years, and now there are several great options. These include shared hosts or virtual private servers running Phusion Passenger (a module for the Apache and Nginx[22] web servers), full-service deployment companies such as Engine Yard and Rails Machine, and cloud deployment services such as Engine Yard Cloud and Heroku.

My favorite Rails deployment option is Heroku, which is a hosted platform built specifically for deploying Rails and other web applications.[23] Heroku makes deploying Rails applications ridiculously easy—as long as your source code is under version control with Git. (This is yet another reason to follow the Git setup steps in Section 1.3 if you haven't already.) The rest of this section is dedicated to deploying our first application to Heroku.

1.4.1 Heroku setup

Heroku uses the PostgreSQL database (pronounced "post-gres-cue-ell", and often called "Postgres" for short), which means that we need to add the `pg` gem in the production environment to allow Rails to talk to Postgres:

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

Appending this code to the **`Gemfile`** from Listing 1.5 yields Listing 1.9.

[21] Though it shouldn't matter for the example applications in the *Rails Tutorial*, if you're worried about accidentally making your app public too soon there are several options; see Section 1.4.4 for one.

[22] Pronounced "Engine X".

[23] Heroku works with any Ruby web platform that uses Rack middleware, which provides a standard interface between web frameworks and web servers. Adoption of the Rack interface has been extraordinarily strong in the Ruby community, including frameworks as varied as Sinatra, Ramaze, Camping, and Rails, which means that Heroku basically supports any Ruby web app.

Listing 1.9. A `Gemfile` with an added `pg` gem for PostgreSQL.

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

gem 'rails', '3.2.11'

group :development do
  gem 'sqlite3', '1.3.5'
end


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

  gem 'uglifier', '1.2.3'
end

gem 'jquery-rails', '2.0.2'

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

To install it, we run `bundle install` with a special flag:

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

The `--without production` option prevents the local installation of any production gems, which in this case is just `pg`.[24]

Next we have to create and configure a new Heroku account. The first step is to sign up for Heroku; after checking your email to complete the creation of your account, install the necessary Heroku software using the Heroku Toolbelt.[25] Then use the `heroku` command to log in at the command line (you may have to exit and restart your terminal program first):

[24] Because the only gem we've added is restricted to a production environment, right now this command doesn't actually install any additional local gems, but for technical reasons it's needed to update `Gemfile.lock`.

[25] https://toolbelt.heroku.com/

```
$ heroku login
```

Finally, navigate back to your Rails project directory and use the `heroku` command to create a place on the Heroku servers for the sample app to live (Listing 1.10).

Listing 1.10. Creating a new application at Heroku.

```
$ cd ~/rails_projects/first_app
$ heroku create
Created http://stormy-cloud-5881.herokuapp.com/ |
git@heroku.com:stormy-cloud-5881.herokuapp.com
Git remote heroku added
```

The `heroku` command creates a new subdomain just for our application, available for immediate viewing. There's nothing there yet, though, so let's get busy deploying.

1.4.2 Heroku deployment, step one

To deploy the application, the first step is to use Git to push it up to Heroku:

```
$ git push heroku master
```

1.4.3 Heroku deployment, step two

There is no step two! We're already done (Figure 1.10). To see your newly deployed application, you can visit the address that you saw when you ran `heroku create` (i.e., Listing 1.10, but with the address for your app, not the address for mine). You can also use an argument to the `heroku` command that automatically opens your browser with the right address:

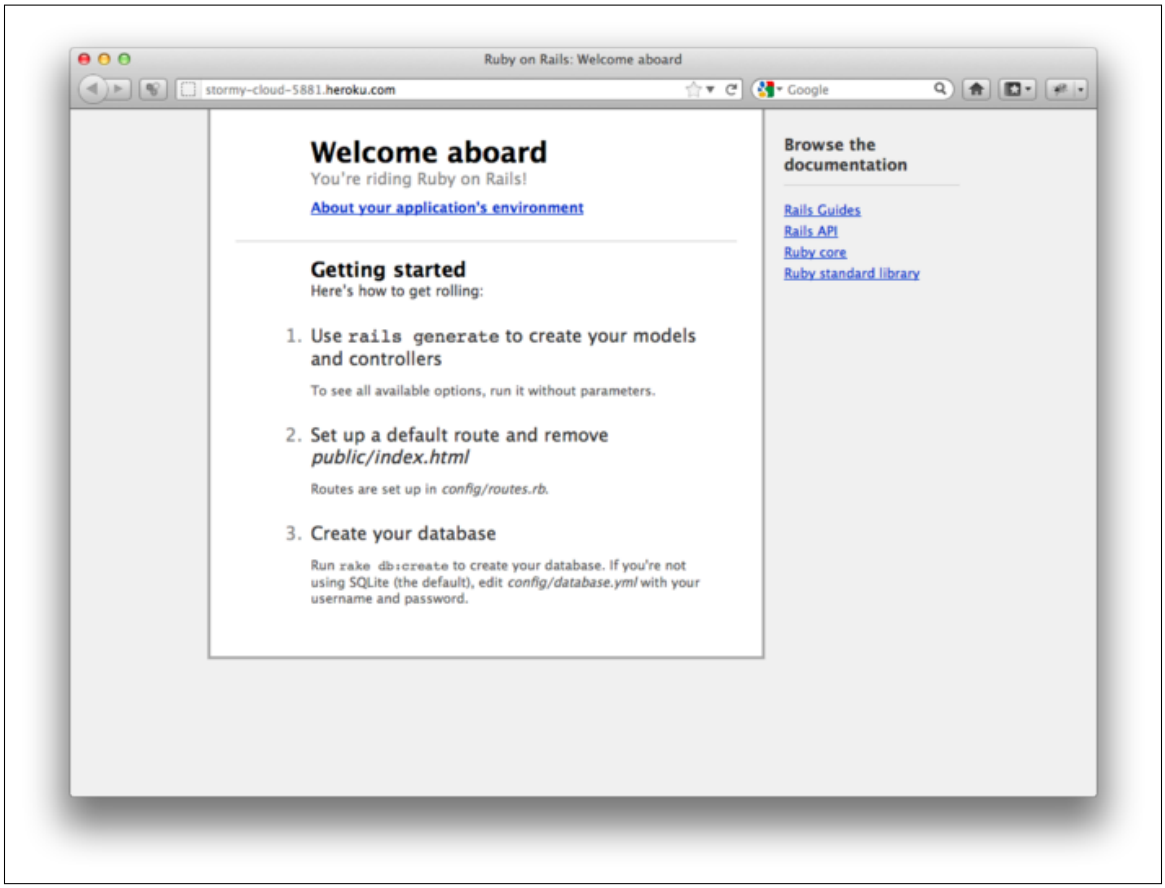

Figure 1.10: The first Rails Tutorial application running on Heroku. (full size)

```
$ heroku open
```

Because of the details of their setup, the "About your application's environment" link doesn't work on Heroku. Don't worry; this is normal. The error will go away (in the context of the full sample application) when we remove the default Rails page in Section 5.3.2.

Once you've deployed successfully, Heroku provides a beautiful interface for administering and configuring your application (Figure 1.11).

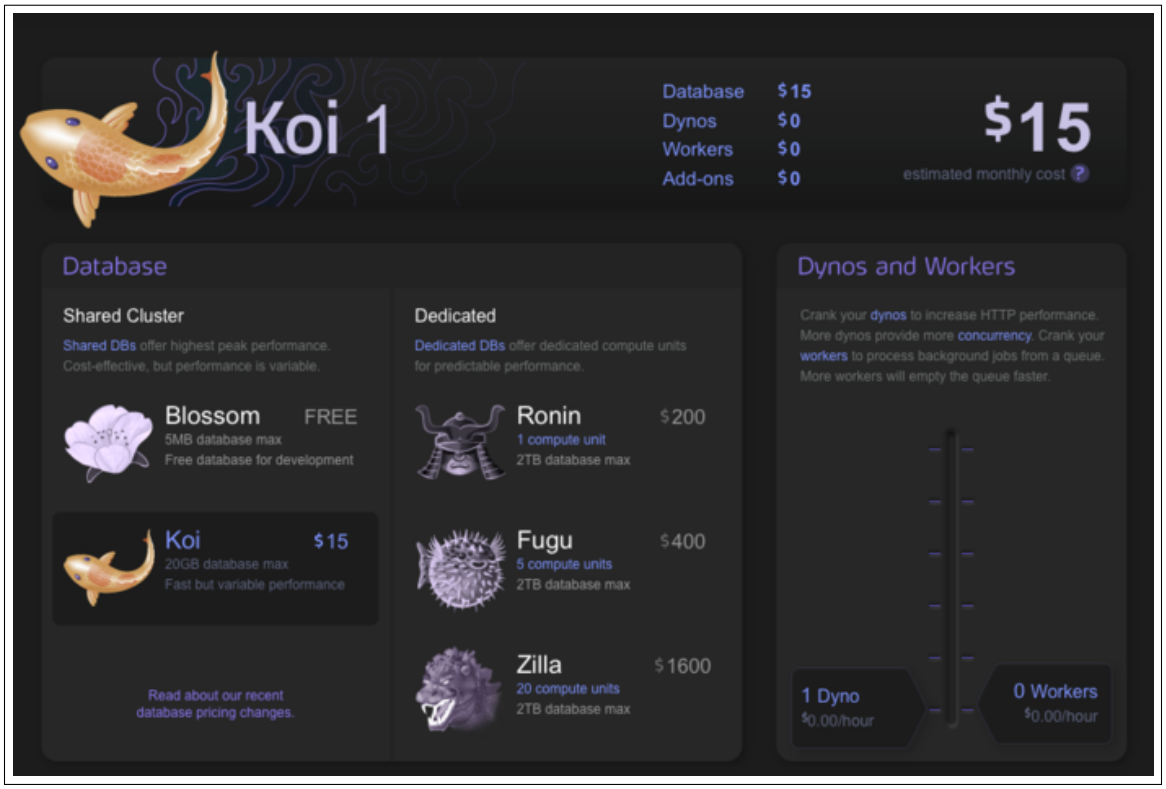

Figure 1.11: The beautiful interface at Heroku. (full size)

1.4.4 Heroku commands

There are many Heroku commands, and we'll barely scratch the surface in this book. Let's take a minute to show just one of them by renaming the application as follows:

```
$ heroku rename railstutorial
```

Don't use this name yourself; it's already taken by me! In fact, you probably shouldn't bother with this step right now; using the default address supplied by Heroku is fine. But if you do want to rename your application, you can arrange for it to be reasonably secure by using a random or obscure subdomain, such as the following:

```
hwpcbmze.heroku.com
seyjhflo.heroku.com
jhyicevg.heroku.com
```

With a random subdomain like this, someone could visit your site only if you gave them the address. (By the way, as a preview of Ruby's compact awesomeness, here's the code I used to generate the random subdomains:

```
('a'..'z').to_a.shuffle[0..7].join
```

Pretty sweet.)

In addition to supporting subdomains, Heroku also supports custom domains. (In fact, the Ruby on Rails Tutorial site lives at Heroku; if you're reading this book online, you're looking at a Heroku-hosted site right now!) See the Heroku documentation for more information about custom domains and other Heroku topics.

1.5 Conclusion

We've come a long way in this chapter: installation, development environment setup, version control, and deployment. If you want to share your progress at this point, feel free to send a tweet or Facebook status update with something like this:

> I'm learning Ruby on Rails with @railstutorial! http://railstutorial.org/

All that's left is to actually start learning Rails! Let's get to it.

Chapter 2

A demo app

In this chapter, we'll develop a simple demonstration application to show off some of the power of Rails. The purpose is to get a high-level overview of Ruby on Rails programming (and web development in general) by rapidly generating an application using *scaffold generators*. As discussed in Box 1.1, the rest of the book will take the opposite approach, developing a full application incrementally and explaining each new concept as it arises, but for a quick overview (and some instant gratification) there is no substitute for scaffolding. The resulting demo app will allow us to interact with it through its URIs, giving us insight into the structure of a Rails application, including a first example of the *REST architecture* favored by Rails.

As with the forthcoming sample application, the demo app will consist of *users* and their associated *microposts* (thus constituting a minimalist Twitter-style app). The functionality will be utterly under-developed, and many of the steps will seem like magic, but worry not: the full sample app will develop a similar application from the ground up starting in Chapter 3, and I will provide plentiful forward-references to later material. In the mean time, have patience and a little faith—the whole point of this tutorial is to take you *beyond* this superficial, scaffold-driven approach to achieve a deeper understanding of Rails.

2.1 Planning the application

In this section, we'll outline our plans for the demo application. As in Section 1.2.3, we'll start by generating the application skeleton using the **rails** command:

```
$ cd ~/rails_projects
$ rails new demo_app
$ cd demo_app
```

Next, we'll use a text editor to update the **Gemfile** needed by Bundler with the contents of Listing 2.1.

Listing 2.1. A **Gemfile** for the demo app.

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

gem 'rails', '3.2.11'

group :development do
  gem 'sqlite3', '1.3.5'
end


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

  gem 'uglifier', '1.2.3'
end

gem 'jquery-rails', '2.0.2'

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

Note that Listing 2.1 is identical to Listing 1.9.

As in Section 1.4.1, we'll install the local gems while suppressing the installation of production gems using the `--without production` option:

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

Finally, we'll put the demo app under version control. Recall that the `rails` command generates a default `.gitignore` file, but depending on your system you may find the augmented file from Listing 1.7 to be more convenient. Then initialize a Git repository and make the first commit:

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

You can also optionally create a new repository (Figure 2.1) and push it up to GitHub:

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

(As with the first app, take care *not* to initialize the GitHub repository with a `README` file.)

Now we're ready to start making the app itself. The typical first step when making a web application is to create a *data model*, which is a representation of the structures needed by our application. In our case, the demo app will be a microblog, with only users and short (micro)posts. Thus, we'll begin with a model for *users* of the app (Section 2.1.1), and then we'll add a model for *microposts* (Section 2.1.2).

2.1.1 Modeling demo users

There are as many choices for a user data model as there are different registration forms on the web; we'll go with a distinctly minimalist approach. Users of our demo app will have a unique `integer` identifier called `id`, a publicly viewable `name` (of type `string`), and an `email` address (also a `string`)

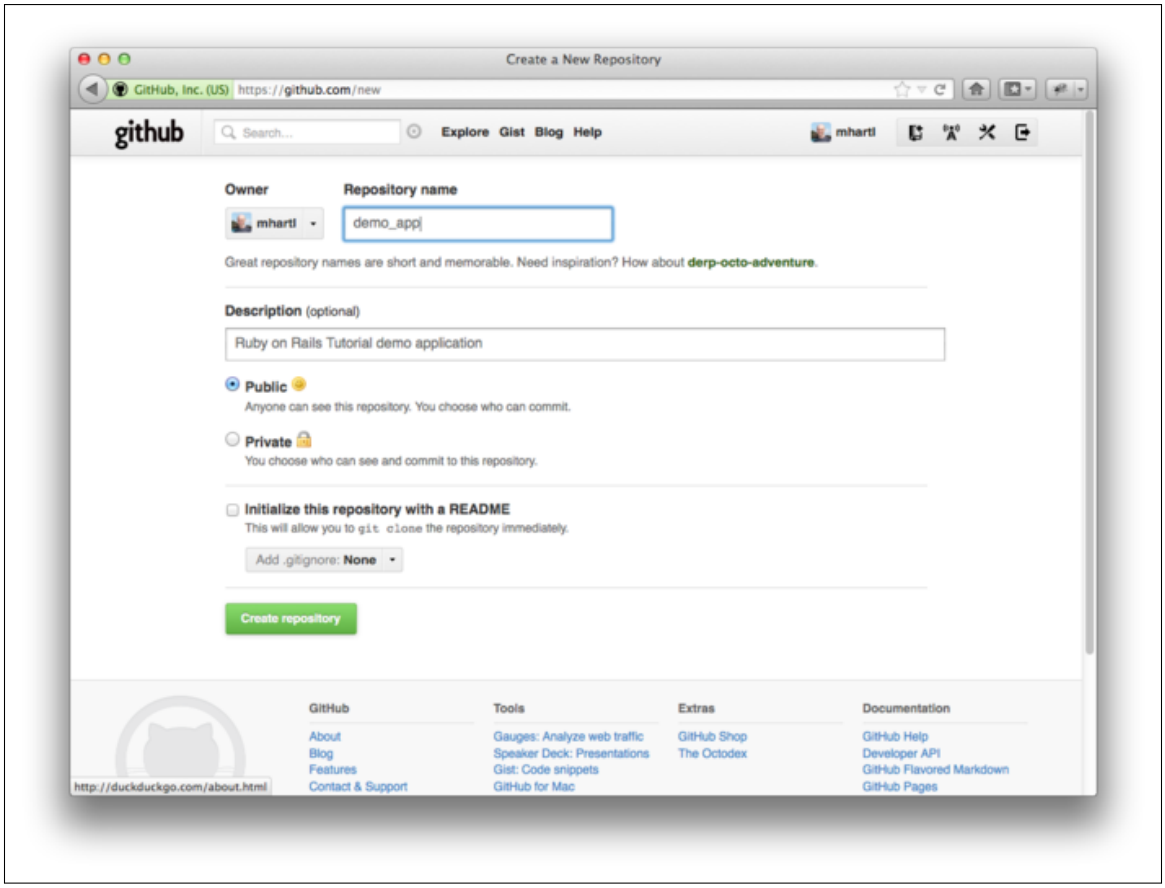

Figure 2.1: Creating a demo app repository at GitHub. (full size)

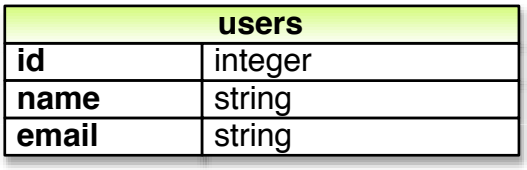

Figure 2.2: The data model for users.

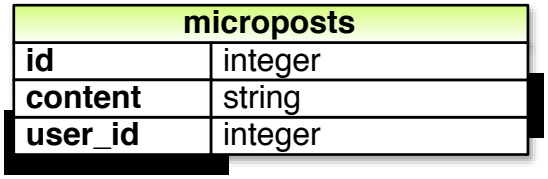

Figure 2.3: The data model for microposts.

that will double as a username. A summary of the data model for users appears in Figure 2.2.

As we'll see starting in Section 6.1.1, the label **users** in Figure 2.2 corresponds to a *table* in a database, and the **id**, **name**, and **email** attributes are *columns* in that table.

2.1.2 Modeling demo microposts

The core of the micropost data model is even simpler than the one for users: a micropost has only an **id** and a **content** field for the micropost's text (of type **string**).[1] There's an additional complication, though: we want to *associate* each micropost with a particular user; we'll accomplish this by recording the **user_id** of the owner of the post. The results are shown in Figure 2.3.

We'll see in Section 2.3.3 (and more fully in Chapter 10) how this **user_-**

[1] When modeling longer posts, such as those for a normal (non-micro) blog, you should use the **text** type in place of **string**.

`id` attribute allows us to succinctly express the notion that a user potentially has many associated microposts.

2.2 The Users resource

In this section, we'll implement the users data model in Section 2.1.1, along with a web interface to that model. The combination will constitute a *Users resource*, which will allow us to think of users as objects that can be created, read, updated, and deleted through the web via the HTTP protocol. As promised in the introduction, our Users resource will be created by a scaffold generator program, which comes standard with each Rails project. I urge you not to look too closely at the generated code; at this stage, it will only serve to confuse you.

Rails scaffolding is generated by passing the `scaffold` command to the `rails generate` script. The argument of the `scaffold` command is the singular version of the resource name (in this case, `User`), together with optional parameters for the data model's attributes:[2]

```
$ rails generate scaffold User name:string email:string
      invoke  active_record
      create    db/migrate/20111123225336_create_users.rb
      create    app/models/user.rb
      invoke    test_unit
      create      test/unit/user_test.rb
      create      test/fixtures/users.yml
       route  resources :users
      invoke  scaffold_controller
      create    app/controllers/users_controller.rb
      invoke    erb
      create      app/views/users
      create      app/views/users/index.html.erb
      create      app/views/users/edit.html.erb
      create      app/views/users/show.html.erb
      create      app/views/users/new.html.erb
      create      app/views/users/_form.html.erb
      invoke    test_unit
      create      test/functional/users_controller_test.rb
      invoke    helper
```

[2]The name of the scaffold follows the convention of *models*, which are singular, rather than resources and controllers, which are plural. Thus, we have `User` instead `Users`.

```
      create      app/helpers/users_helper.rb
      invoke      test_unit
      create        test/unit/helpers/users_helper_test.rb
      invoke  assets
      invoke    coffee
      create      app/assets/javascripts/users.js.coffee
      invoke    scss
      create      app/assets/stylesheets/users.css.scss
      invoke  scss
      create    app/assets/stylesheets/scaffolds.css.scss
```

By including **`name:string`** and **`email:string`**, we have arranged for the User model to have the form shown in Figure 2.2. (Note that there is no need to include a parameter for **`id`**; it is created automatically by Rails for use as the *primary key* in the database.)

To proceed with the demo application, we first need to *migrate* the database using *Rake* (Box 2.1):

```
$ bundle exec rake db:migrate
==  CreateUsers: migrating ====================================================
-- create_table(:users)
   -> 0.0017s
==  CreateUsers: migrated (0.0018s) ===========================================
```

This simply updates the database with our new **`users`** data model. (We'll learn more about database migrations starting in Section 6.1.1.) Note that, in order to ensure that the command uses the version of Rake corresponding to our **`Gemfile`**, we need to run **`rake`** using **`bundle exec`**.

With that, we can run the local web server using **`rails s`**, which is a shortcut for **`rails server`**:

```
$ rails s
```

Now the demo application should be ready to go at http://localhost:3000/.

Box 2.1. Rake

In the Unix tradition, the *make* utility has played an important role in building executable programs from source code; many a computer hacker has committed to muscle memory the line

```
$ ./configure && make && sudo make install
```

commonly used to compile code on Unix systems (including Linux and Mac OS X).

Rake is *Ruby make*, a make-like language written in Ruby. Rails uses Rake extensively, especially for the innumerable little administrative tasks necessary when developing database-backed web applications. The **`rake db:migrate`** command is probably the most common, but there are many others; you can see a list of database tasks using **`-T db`**:

```
$ bundle exec rake -T db
```

To see all the Rake tasks available, run

```
$ bundle exec rake -T
```

The list is likely to be overwhelming, but don't worry, you don't have to know all (or even most) of these commands. By the end of the *Rails Tutorial*, you'll know all the most important ones.

2.2.1 A user tour

Visiting the root url http://localhost:3000/ shows the same default Rails page shown in Figure 1.3, but in generating the Users resource scaffolding we have also created a large number of pages for manipulating users. For example, the page for listing all users is at /users, and the page for making a new user is

URI	Action	Purpose
/users	`index`	page to list all users
/users/1	`show`	page to show user with id `1`
/users/new	`new`	page to make a new user
/users/1/edit	`edit`	page to edit user with id `1`

Table 2.1: The correspondence between pages and URIs for the Users resource.

at /users/new. The rest of this section is dedicated to taking a whirlwind tour through these user pages. As we proceed, it may help to refer to Table 2.1, which shows the correspondence between pages and URIs.

We start with the page to show all the users in our application, called `index`; as you might expect, initially there are no users at all (Figure 2.4).

To make a new user, we visit the `new` page, as shown in Figure 2.5. (Since the http://localhost:3000 part of the address is implicit whenever we are developing locally, I'll usually omit it from now on.) In Chapter 7, this will become the user signup page.

We can create a user by entering name and email values in the text fields and then clicking the Create User button. The result is the user `show` page, as seen in Figure 2.6. (The green welcome message is accomplished using the *flash*, which we'll learn about in Section 7.4.2.) Note that the URI is /users/1; as you might suspect, the number `1` is simply the user's `id` attribute from Figure 2.2. In Section 7.1, this page will become the user's profile.

To change a user's information, we visit the `edit` page (Figure 2.7). By modifying the user information and clicking the Update User button, we arrange to change the information for the user in the demo application (Figure 2.8). (As we'll see in detail starting in Chapter 6, this user data is stored in a database back-end.) We'll add user edit/update functionality to the sample application in Section 9.1.

Now we'll create a second user by revisiting the `new` page and submitting a second set of user information; the resulting user `index` is shown in Figure 2.9. Section 7.1 will develop the user index into a more polished page for showing all users.

Having shown how to create, show, and edit users, we come finally to destroying them (Figure 2.10). You should verify that clicking on the link in

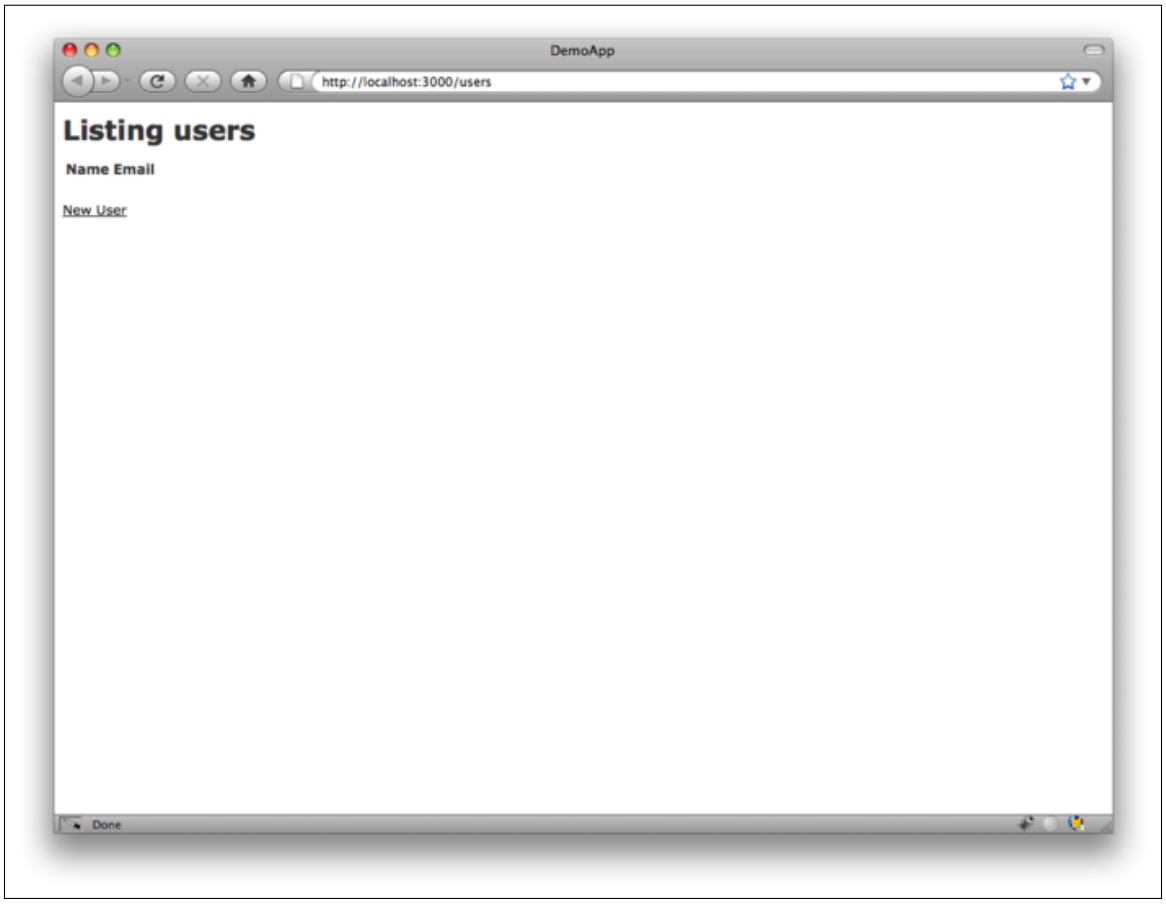

Figure 2.4: The initial index page for the Users resource (/users). (full size)

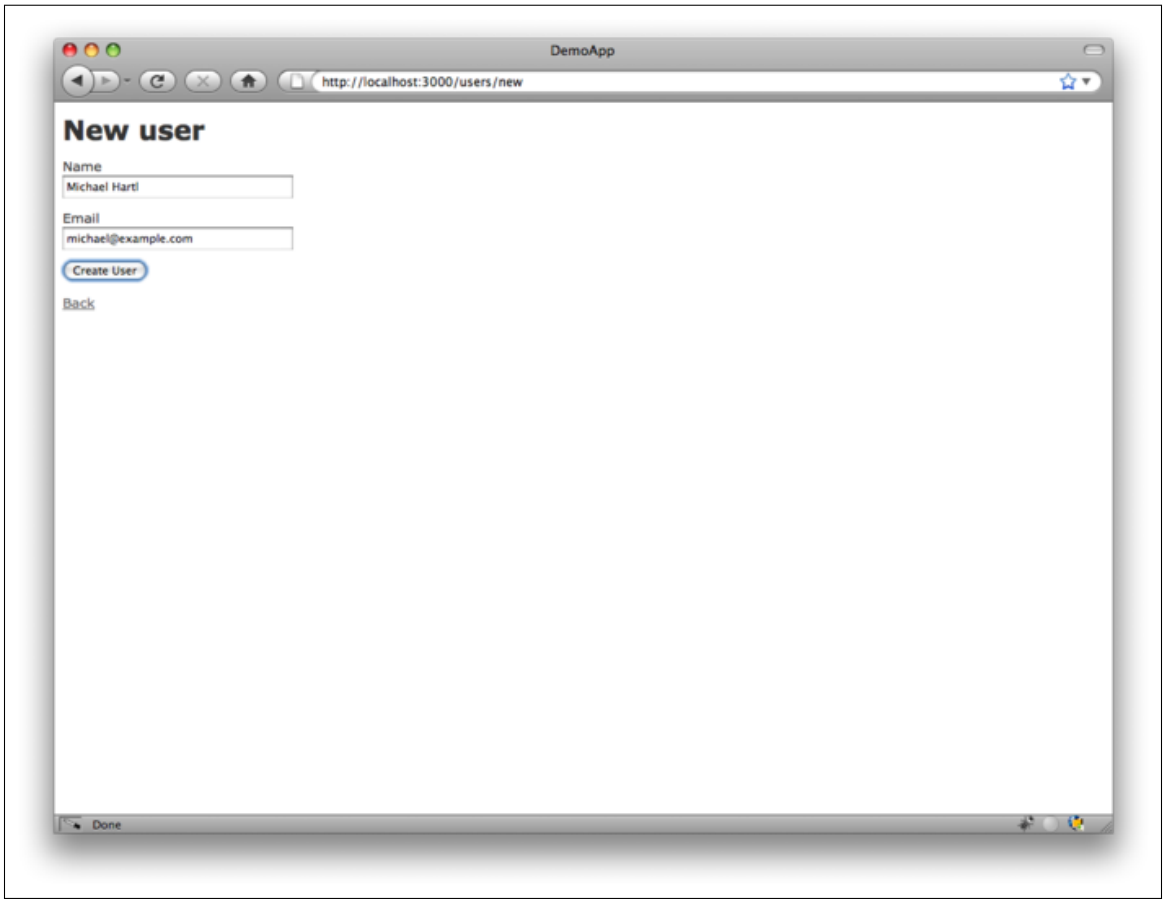

Figure 2.5: The new user page (/users/new). (full size)

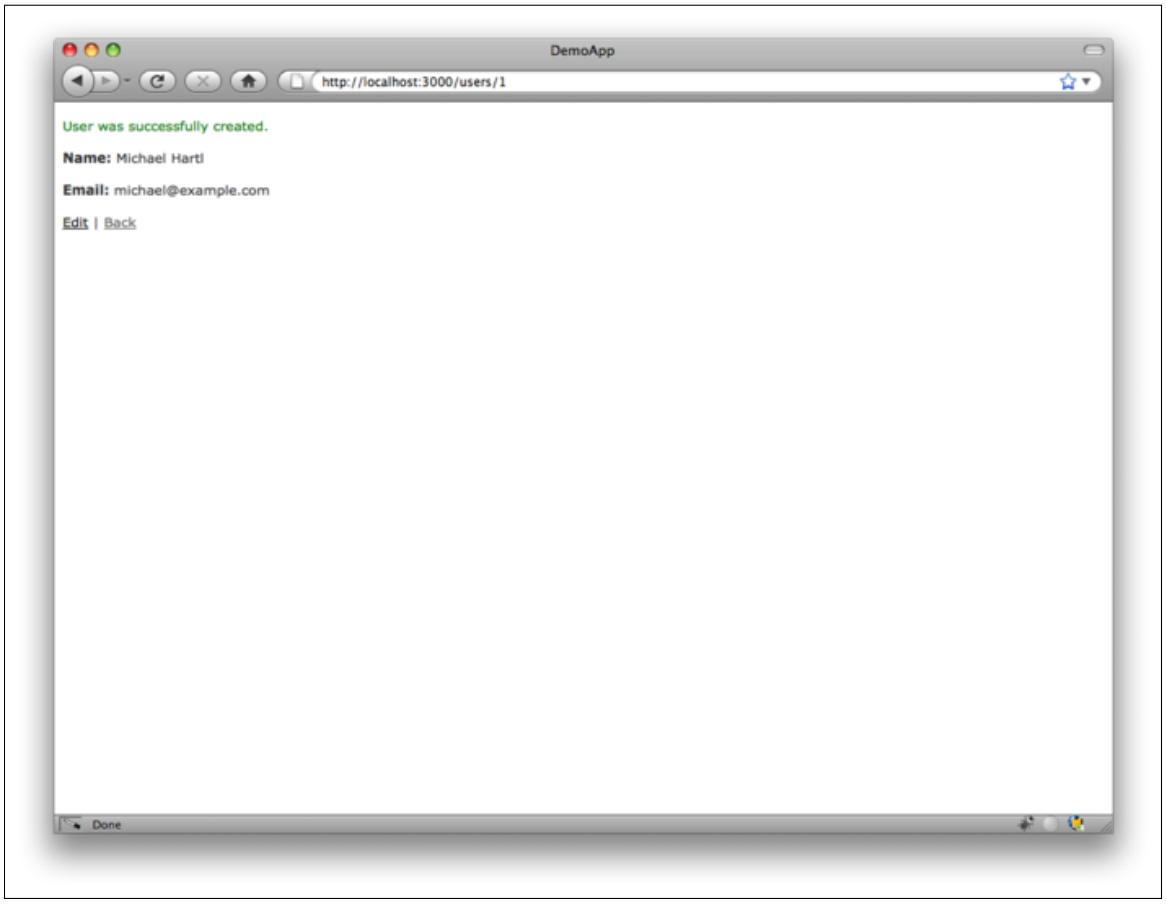

Figure 2.6: The page to show a user (/users/1). (full size)

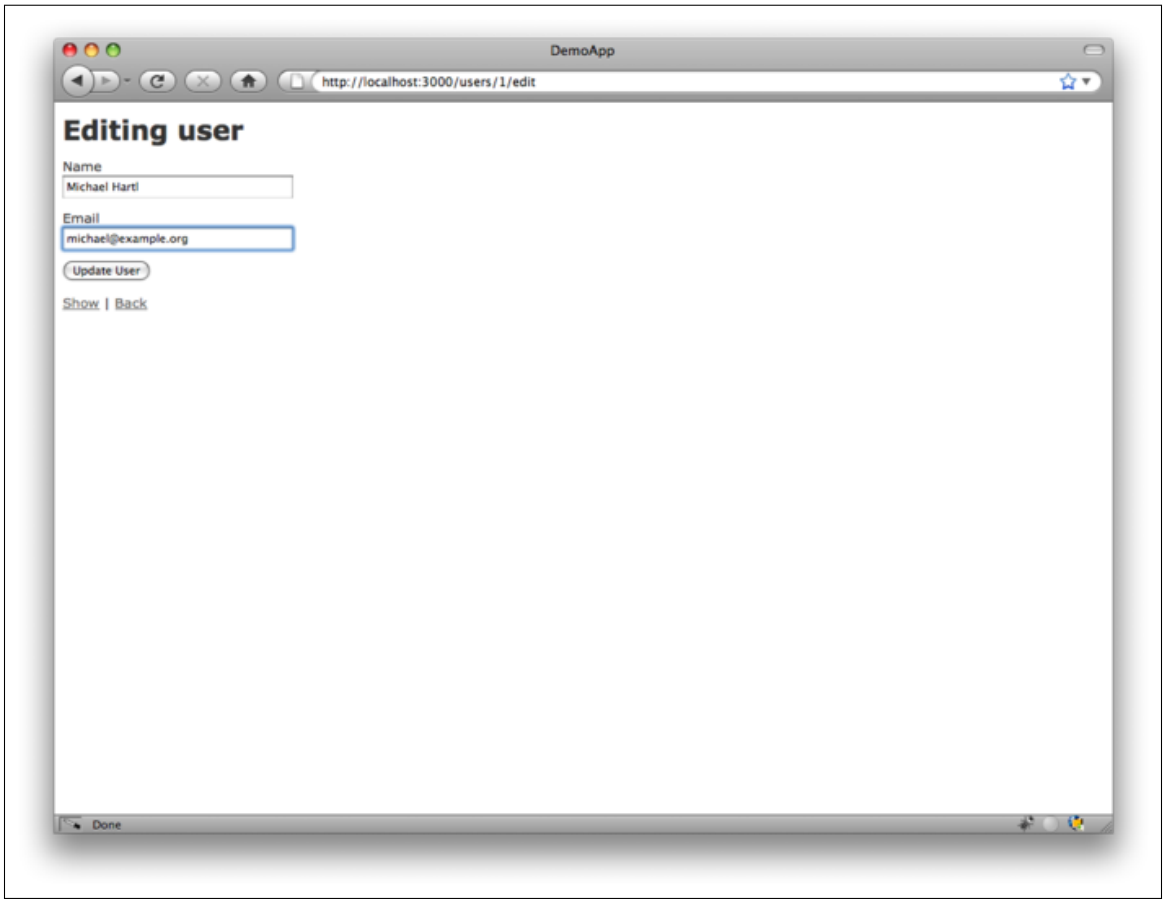

Figure 2.7: The user edit page (/users/1/edit). (full size)

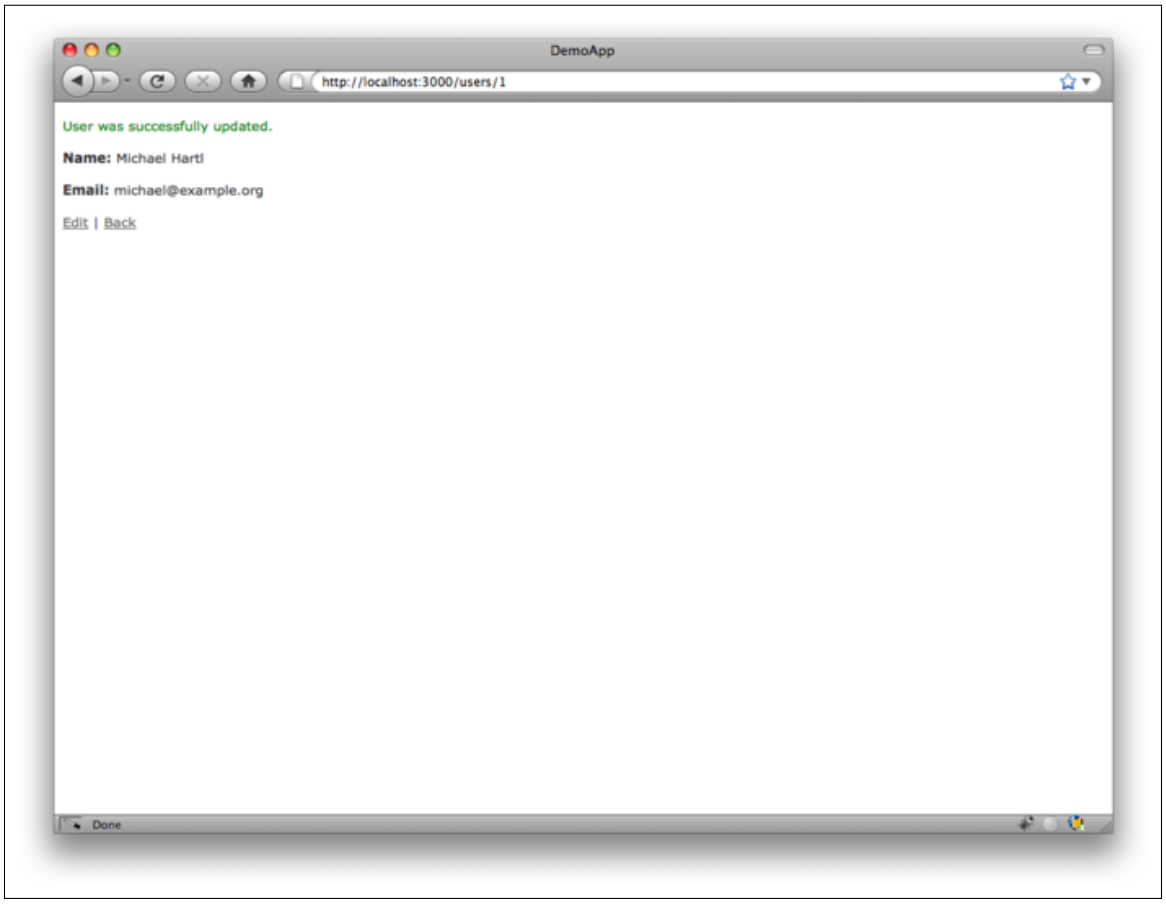

Figure 2.8: A user with updated information. (full size)

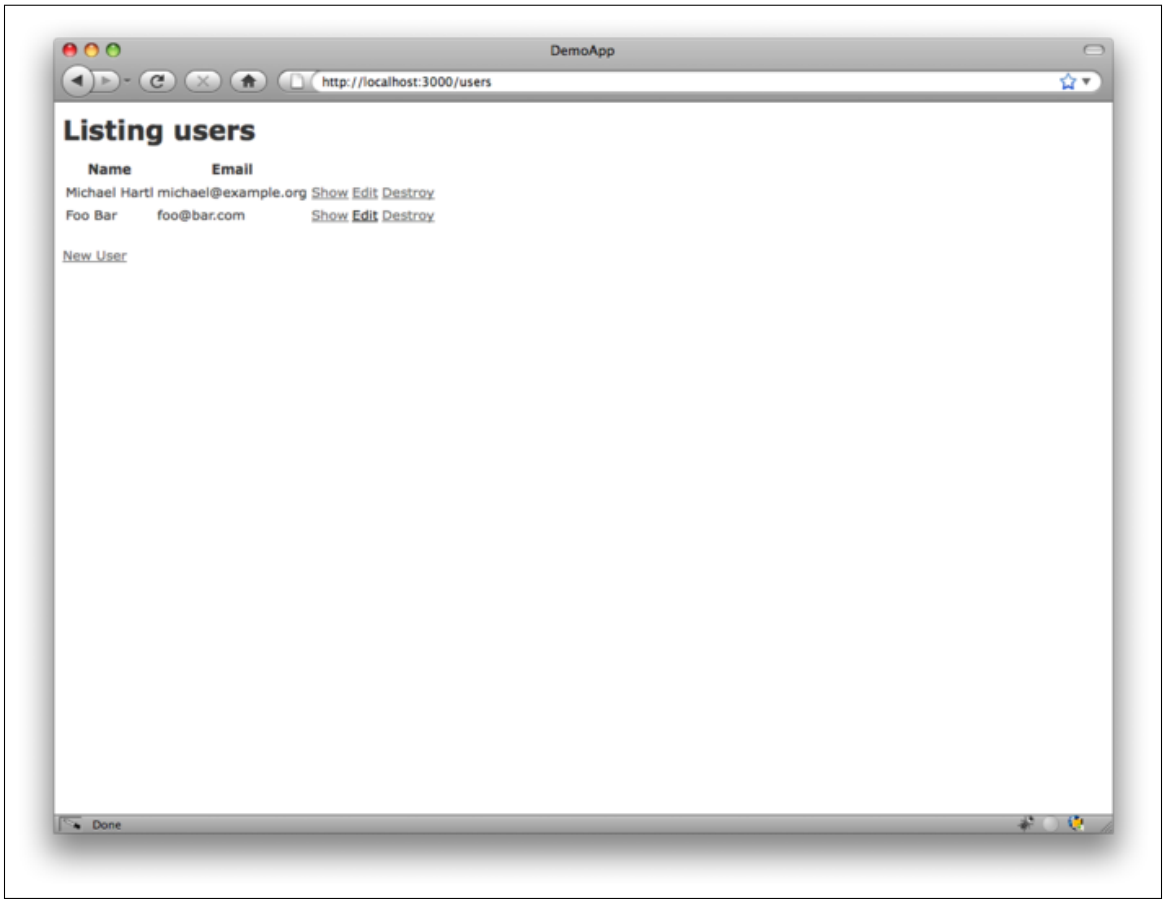

Figure 2.9: The user index page (/users) with a second user. (full size)

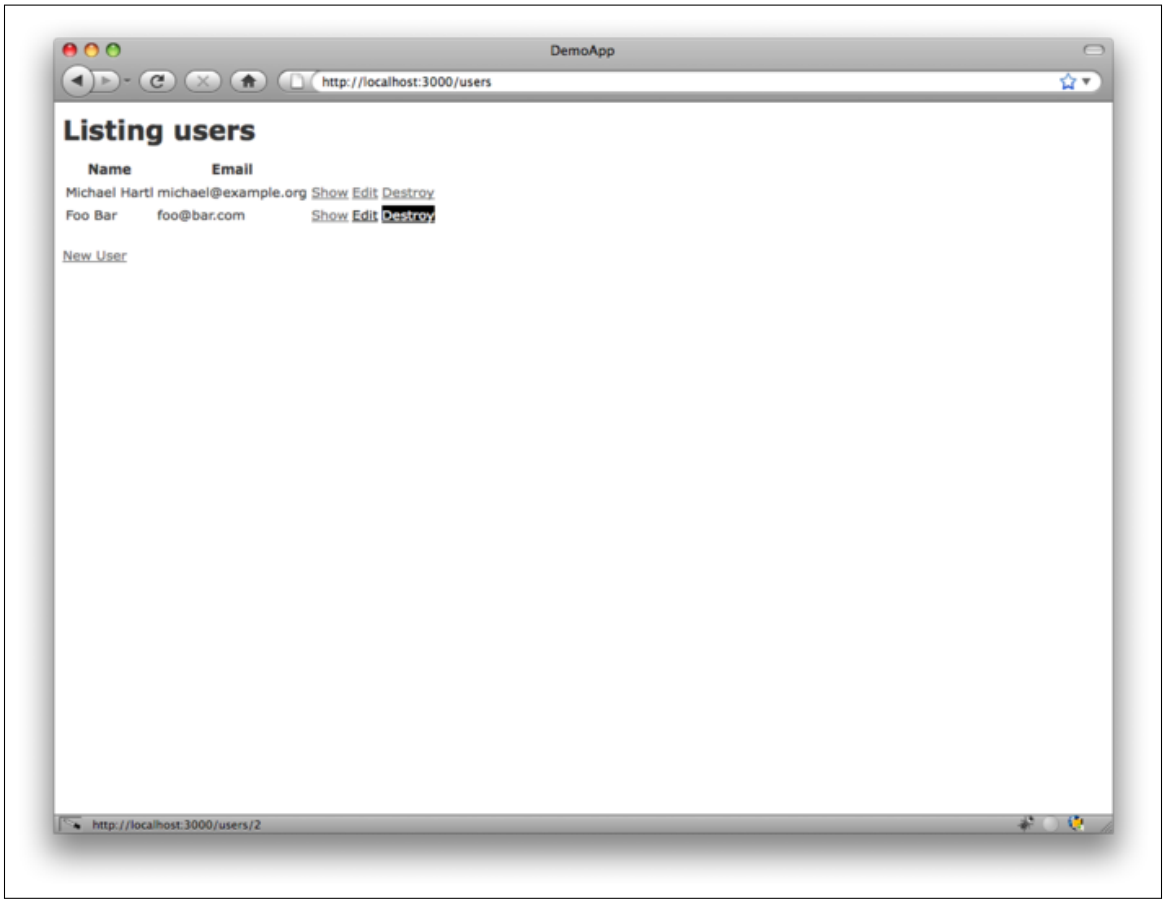

Figure 2.10: Destroying a user. (full size)

Figure 2.10 destroys the second user, yielding an index page with only one user. (If it doesn't work, be sure that JavaScript is enabled in your browser; Rails uses JavaScript to issue the request needed to destroy a user.) Section 9.4 adds user deletion to the sample app, taking care to restrict its use to a special class of administrative users.

2.2.2 MVC in action

Now that we've completed a quick overview of the Users resource, let's examine one particular part of it in the context of the Model-View-Controller (MVC)

pattern introduced in Section 1.2.6. Our strategy will be to describe the results of a typical browser hit—a visit to the user index page at /users—in terms of MVC (Figure 2.11).

1. The browser issues a request for the /users URI.
2. Rails routes /users to the **`index`** action in the Users controller.
3. The **`index`** action asks the User model to retrieve all users (**`User.all`**).
4. The User model pulls all the users from the database.
5. The User model returns the list of users to the controller.
6. The controller captures the users in the **`@users`** variable, which is passed to the **`index`** view.
7. The view uses embedded Ruby to render the page as HTML.
8. The controller passes the HTML back to the browser.[3]

We start with a request issued from the browser—i.e., the result of typing a URI in the address bar or clicking on a link (Step 1 in Figure 2.11). This request hits the *Rails router* (Step 2), which dispatches to the proper *controller action* based on the URI (and, as we'll see in Box 3.2, the type of request). The code to create the mapping of user URIs to controller actions for the Users resource appears in Listing 2.2; this code effectively sets up the table of URI/action pairs seen in Table 2.1. (The strange notation **`:users`** is a *symbol*, which we'll learn about in Section 4.3.3.)

Listing 2.2. The Rails routes, with a rule for the Users resource.
`config/routes.rb`

[3] Some references indicate that the view returns the HTML directly to the browser (via a web server such as Apache or Nginx). Regardless of the implementation details, I prefer to think of the controller as a central hub through which all the application's information flows.

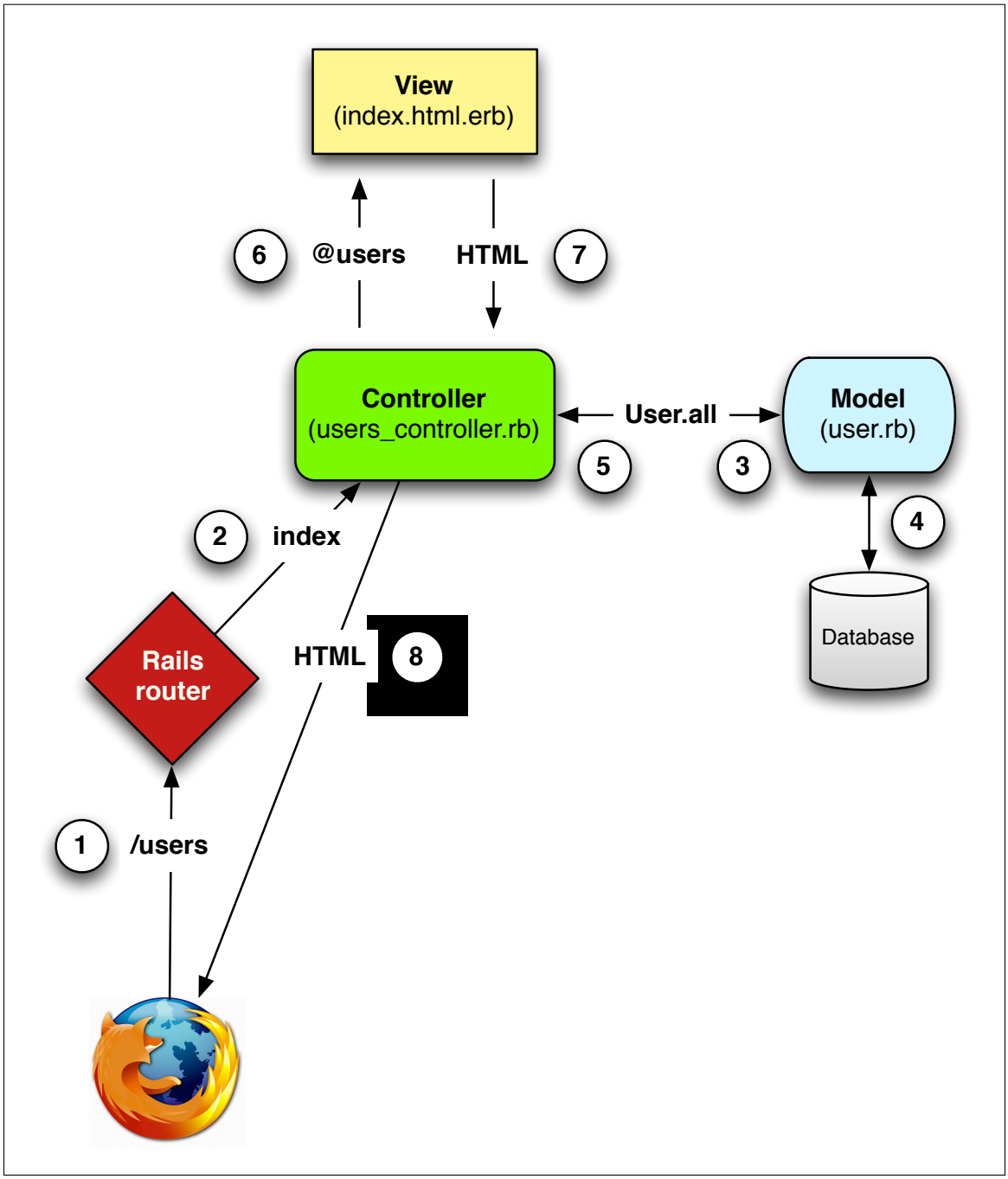

Figure 2.11: A detailed diagram of MVC in Rails. (full size)

```
DemoApp::Application.routes.draw do
  resources :users
  .
  .
  .
end
```

The pages from the tour in Section 2.2.1 correspond to *actions* in the Users *controller*, which is a collection of related actions; the controller generated by the scaffolding is shown schematically in Listing 2.3. Note the notation `class UsersController < ApplicationController`; this is an example of a Ruby *class* with *inheritance*. (We'll discuss inheritance briefly in Section 2.3.4 and cover both subjects in more detail in Section 4.4.)

Listing 2.3. The Users controller in schematic form.
app/controllers/users_controller.rb

```
class UsersController < ApplicationController

  def index
    .
    .
    .
  end

  def show
    .
    .
    .
  end

  def new
    .
    .
    .
  end

  def create
    .
    .
    .
  end
```

```
  def edit
    .
    .
    .
  end

  def update
    .
    .
    .
  end

  def destroy
    .
    .
    .
  end
end
```

You may notice that there are more actions than there are pages; the **index**, **show**, **new**, and **edit** actions all correspond to pages from Section 2.2.1, but there are additional **create**, **update**, and **destroy** actions as well. These actions don't typically render pages (although they sometimes do); instead, their main purpose is to modify information about users in the database. This full suite of controller actions, summarized in Table 2.2, represents the implementation of the REST architecture in Rails (Box 2.2), which is based on the ideas of *representational state transfer* identified and named by computer scientist Roy Fielding.[4] Note from Table 2.2 that there is some overlap in the URIs; for example, both the user **show** action and the **update** action correspond to the URI /users/1. The difference between them is the HTTP request method they respond to. We'll learn more about HTTP request methods starting in Section 3.2.1.

Box 2.2. REpresentational State Transfer (REST)

If you read much about Ruby on Rails web development, you'll see a lot of references to "REST", which is an acronym for REpresentational State Trans-

[4] Fielding, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine, 2000.

HTTP request	URI	Action	Purpose
GET	/users	**index**	page to list all users
GET	/users/1	**show**	page to show user with id **1**
GET	/users/new	**new**	page to make a new user
POST	/users	**create**	create a new user
GET	/users/1/edit	**edit**	page to edit user with id **1**
PUT	/users/1	**update**	update user with id **1**
DELETE	/users/1	**destroy**	delete user with id **1**

Table 2.2: RESTful routes provided by the Users resource in Listing 2.2.

fer. REST is an architectural style for developing distributed, networked systems and software applications such as the World Wide Web and web applications. Although REST theory is rather abstract, in the context of Rails applications REST means that most application components (such as users and microposts) are modeled as *resources* that can be created, read, updated, and deleted—operations that correspond both to the CRUD operations of relational databases and the four fundamental HTTP request methods: `POST`, `GET`, `PUT`, and `DELETE`. (We'll learn more about HTTP requests in Section 3.2.1 and especially Box 3.2.)

As a Rails application developer, the RESTful style of development helps you make choices about which controllers and actions to write: you simply structure the application using resources that get created, read, updated, and deleted. In the case of users and microposts, this process is straightforward, since they are naturally resources in their own right. In Chapter 11, we'll see an example where REST principles allow us to model a subtler problem, "following users", in a natural and convenient way.

To examine the relationship between the Users controller and the User model, let's focus on a simplified version of the **index** action, shown in Listing 2.4. (The scaffold code is ugly and confusing, so I've suppressed it.)

Listing 2.4. The simplified user **index** action for the demo application.
app/controllers/users_controller.rb

```
class UsersController < ApplicationController

  def index
    @users = User.all
  end
  .
  .
  .
end
```

This **index** action has the line **@users = User.all** (Step 3), which asks the User model to retrieve a list of all the users from the database (Step 4), and then places them in the variable **@users** (pronounced "at-users") (Step 5). The User model itself appears in Listing 2.5; although it is rather plain, it comes equipped with a large amount of functionality because of inheritance (Section 2.3.4 and Section 4.4). In particular, by using the Rails library called *Active Record*, the code in Listing 2.5 arranges for **User.all** to return all the users. (We'll learn about the **attr_accessible** line in Section 6.1.2. *Note*: This line will not appear if you are using Rails 3.2.2 or earlier.)

Listing 2.5. The User model for the demo application.
app/models/user.rb

```
class User < ActiveRecord::Base
  attr_accessible :email, :name
end
```

Once the **@users** variable is defined, the controller calls the *view* (Step 6), shown in Listing 2.6. Variables that start with the **@** sign, called *instance variables*, are automatically available in the view; in this case, the **index.html.erb** view in Listing 2.6 iterates through the **@users** list and outputs a line of HTML for each one. (Remember, you aren't supposed to understand this code right now. It is shown only for purposes of illustration.)

Listing 2.6. The view for the user index.
app/views/users/index.html.erb

```
<h1>Listing users</h1>

<table>
  <tr>
    <th>Name</th>
    <th>Email</th>
    <th></th>
    <th></th>
    <th></th>
  </tr>

<% @users.each do |user| %>
  <tr>
    <td><%= user.name %></td>
    <td><%= user.email %></td>
    <td><%= link_to 'Show', user %></td>
    <td><%= link_to 'Edit', edit_user_path(user) %></td>
    <td><%= link_to 'Destroy', user, method: :delete,
                                     data: { confirm: 'Are you sure?' } %></td>
  </tr>
<% end %>
</table>

<br />

<%= link_to 'New User', new_user_path %>
```

The view converts its contents to HTML (Step 7), which is then returned by the controller to the browser for display (Step 8).

2.2.3 Weaknesses of this Users resource

Though good for getting a general overview of Rails, the scaffold Users resource suffers from a number of severe weaknesses.

- **No data validations.** Our User model accepts data such as blank names and invalid email addresses without complaint.

- **No authentication.** We have no notion signing in or out, and no way to prevent any user from performing any operation.

- **No tests.** This isn't technically true—the scaffolding includes rudimentary tests—but the generated tests are ugly and inflexible, and they don't test for data validation, authentication, or any other custom requirements.
- **No layout.** There is no consistent site styling or navigation.
- **No real understanding.** If you understand the scaffold code, you probably shouldn't be reading this book.

2.3 The Microposts resource

Having generated and explored the Users resource, we turn now to the associated Microposts resource. Throughout this section, I recommend comparing the elements of the Microposts resource with the analogous user elements from Section 2.2; you should see that the two resources parallel each other in many ways. The RESTful structure of Rails applications is best absorbed by this sort of repetition of form; indeed, seeing the parallel structure of Users and Microposts even at this early stage is one of the prime motivations for this chapter. (As we'll see, writing applications more robust than the toy example in this chapter takes considerable effort—we won't see the Microposts resource again until Chapter 10—and I didn't want to defer its first appearance quite that far.)

2.3.1 A micropost microtour

As with the Users resource, we'll generate scaffold code for the Microposts resource using **`rails generate scaffold`**, in this case implementing the data model from Figure 2.3:[5]

```
$ rails generate scaffold Micropost content:string user_id:integer
      invoke  active_record
      create    db/migrate/20111123225811_create_microposts.rb
      create    app/models/micropost.rb
```

[5] As with the User scaffold, the scaffold generator for microposts follows the singular convention of Rails models; thus, we have **`generate Micropost`**.

```
      invoke    test_unit
      create      test/unit/micropost_test.rb
      create      test/fixtures/microposts.yml
       route  resources :microposts
      invoke  scaffold_controller
      create    app/controllers/microposts_controller.rb
      invoke    erb
      create      app/views/microposts
      create      app/views/microposts/index.html.erb
      create      app/views/microposts/edit.html.erb
      create      app/views/microposts/show.html.erb
      create      app/views/microposts/new.html.erb
      create      app/views/microposts/_form.html.erb
      invoke    test_unit
      create      test/functional/microposts_controller_test.rb
      invoke    helper
      create      app/helpers/microposts_helper.rb
      invoke      test_unit
      create        test/unit/helpers/microposts_helper_test.rb
      invoke  assets
      invoke    coffee
      create      app/assets/javascripts/microposts.js.coffee
      invoke    scss
      create      app/assets/stylesheets/microposts.css.scss
      invoke  scss
   identical    app/assets/stylesheets/scaffolds.css.scss
```

To update our database with the new data model, we need to run a migration as in Section 2.2:

```
$ bundle exec rake db:migrate
==  CreateMicroposts: migrating ===============================================
-- create_table(:microposts)
   -> 0.0023s
==  CreateMicroposts: migrated (0.0026s) ======================================
```

Now we are in a position to create microposts in the same way we created users in Section 2.2.1. As you might guess, the scaffold generator has updated the Rails routes file with a rule for Microposts resource, as seen in Listing 2.7.[6] As with users, the `resources :microposts` routing rule maps micropost URIs to actions in the Microposts controller, as seen in Table 2.3.

[6]The scaffold code may have extra newlines compared to Listing 2.7. This is not a cause for concern, as Ruby ignores extra newlines.

HTTP request	URI	Action	Purpose
GET	/microposts	**index**	page to list all microposts
GET	/microposts/1	**show**	page to show micropost with id **1**
GET	/microposts/new	**new**	page to make a new micropost
POST	/microposts	**create**	create a new micropost
GET	/microposts/1/edit	**edit**	page to edit micropost with id **1**
PUT	/microposts/1	**update**	update micropost with id **1**
DELETE	/microposts/1	**destroy**	delete micropost with id **1**

Table 2.3: RESTful routes provided by the Microposts resource in Listing 2.7.

Listing 2.7. The Rails routes, with a new rule for Microposts resources.
config/routes.rb

```
DemoApp::Application.routes.draw do
  resources :microposts
  resources :users
  .
  .
  .
end
```

The Microposts controller itself appears in schematic form Listing 2.8. Note that, apart from having **MicropostsController** in place of **UsersController**, Listing 2.8 is *identical* to the code in Listing 2.3. This is a reflection of the REST architecture common to both resources.

Listing 2.8. The Microposts controller in schematic form.
app/controllers/microposts_controller.rb

```
class MicropostsController < ApplicationController

  def index
    .
    .
    .
  end

  def show
    .
```

```
    .
    .
  end

  def new
    .
    .
    .
  end

  def create
    .
    .
    .
  end

  def edit
    .
    .
    .
  end

  def update
    .
    .
    .
  end

  def destroy
    .
    .
    .
  end
end
```

To make some actual microposts, we enter information at the new microposts page, /microposts/new, as seen in Figure 2.12.

At this point, go ahead and create a micropost or two, taking care to make sure that at least one has a `user_id` of `1` to match the id of the first user created in Section 2.2.1. The result should look something like Figure 2.13.

2.3.2 Putting the *micro* in microposts

Any *micro*post worthy of the name should have some means of enforcing the length of the post. Implementing this constraint in Rails is easy with *vali-*

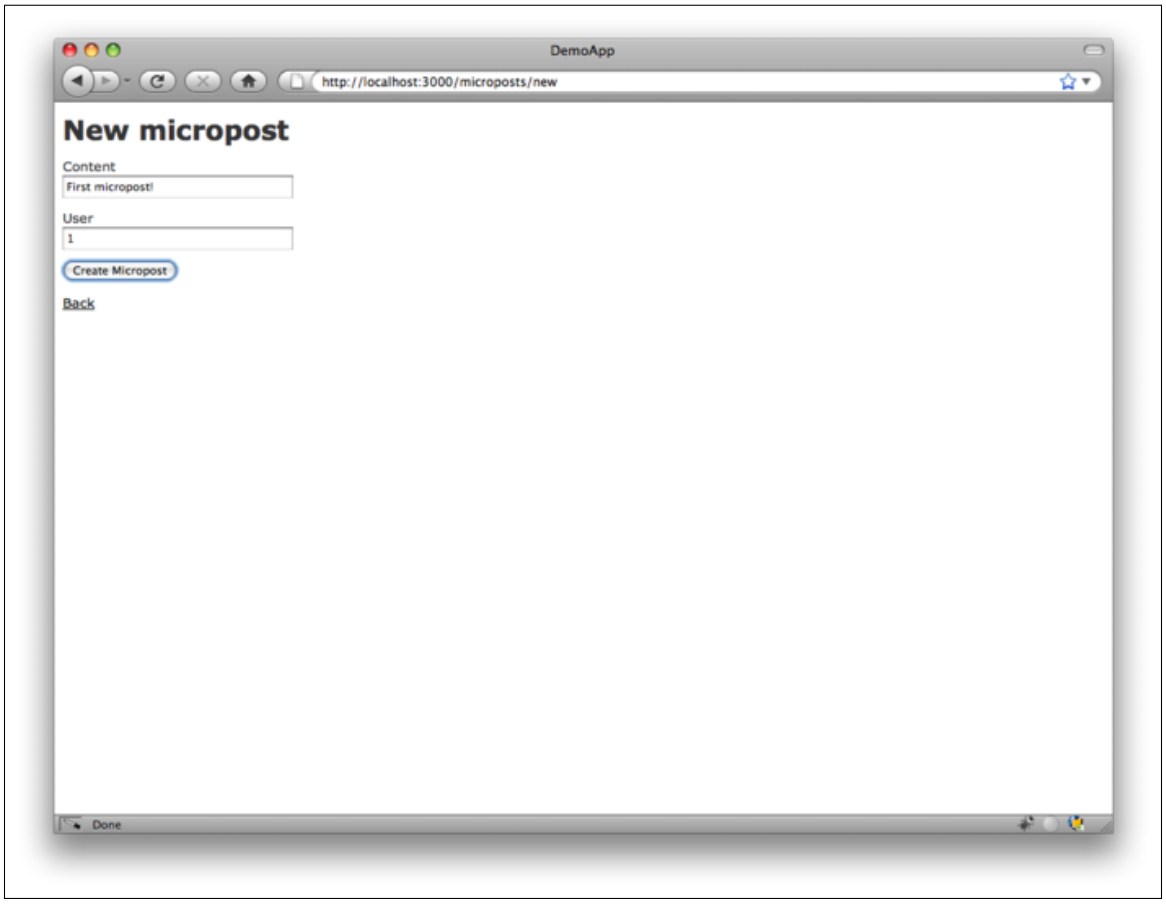

Figure 2.12: The new micropost page (/microposts/new). (full size)

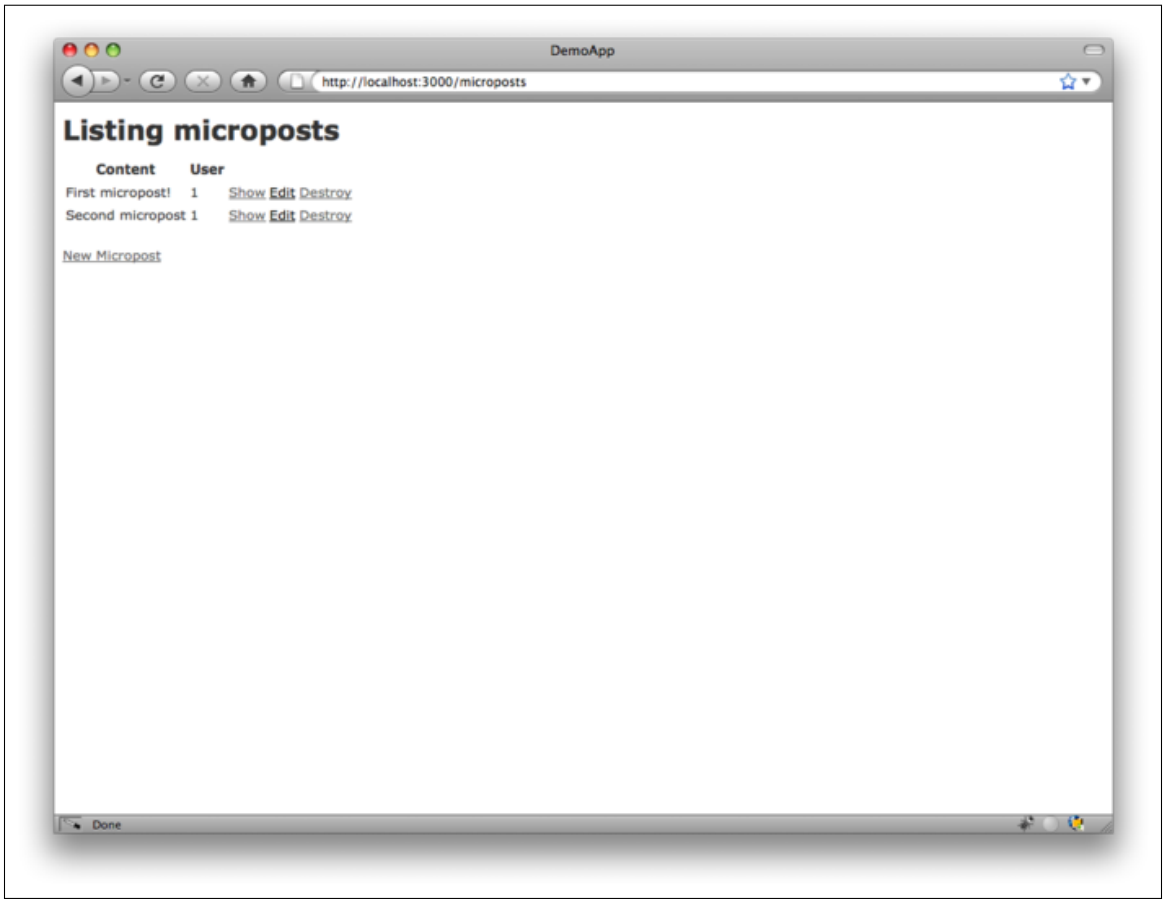

Figure 2.13: The micropost index page (/microposts). (full size)

dations; to accept microposts with at most 140 characters (à la Twitter), we use a *length* validation. At this point, you should open the file **app/models/micropost.rb** in your text editor or IDE and fill it with the contents of Listing 2.9. (The use of **validates** in Listing 2.9 is characteristic of Rails 3; if you've previously worked with Rails 2.3, you should compare this to the use of **validates_length_of**.)

Listing 2.9. Constraining microposts to be at most 140 characters.
app/models/micropost.rb

```
class Micropost < ActiveRecord::Base
  attr_accessible :content, :user_id
  validates :content, :length => { :maximum => 140 }
end
```

The code in Listing 2.9 may look rather mysterious—we'll cover validations more thoroughly starting in Section 6.2—but its effects are readily apparent if we go to the new micropost page and enter more than 140 characters for the content of the post. As seen in Figure 2.14, Rails renders *error messages* indicating that the micropost's content is too long. (We'll learn more about error messages in Section 7.3.2.)

2.3.3 A user `has_many` microposts

One of the most powerful features of Rails is the ability to form *associations* between different data models. In the case of our User model, each user potentially has many microposts. We can express this in code by updating the User and Micropost models as in Listing 2.10 and Listing 2.11.

Listing 2.10. A user has many microposts.
app/models/user.rb

```
class User < ActiveRecord::Base
  attr_accessible :email, :name
  has_many :microposts
end
```

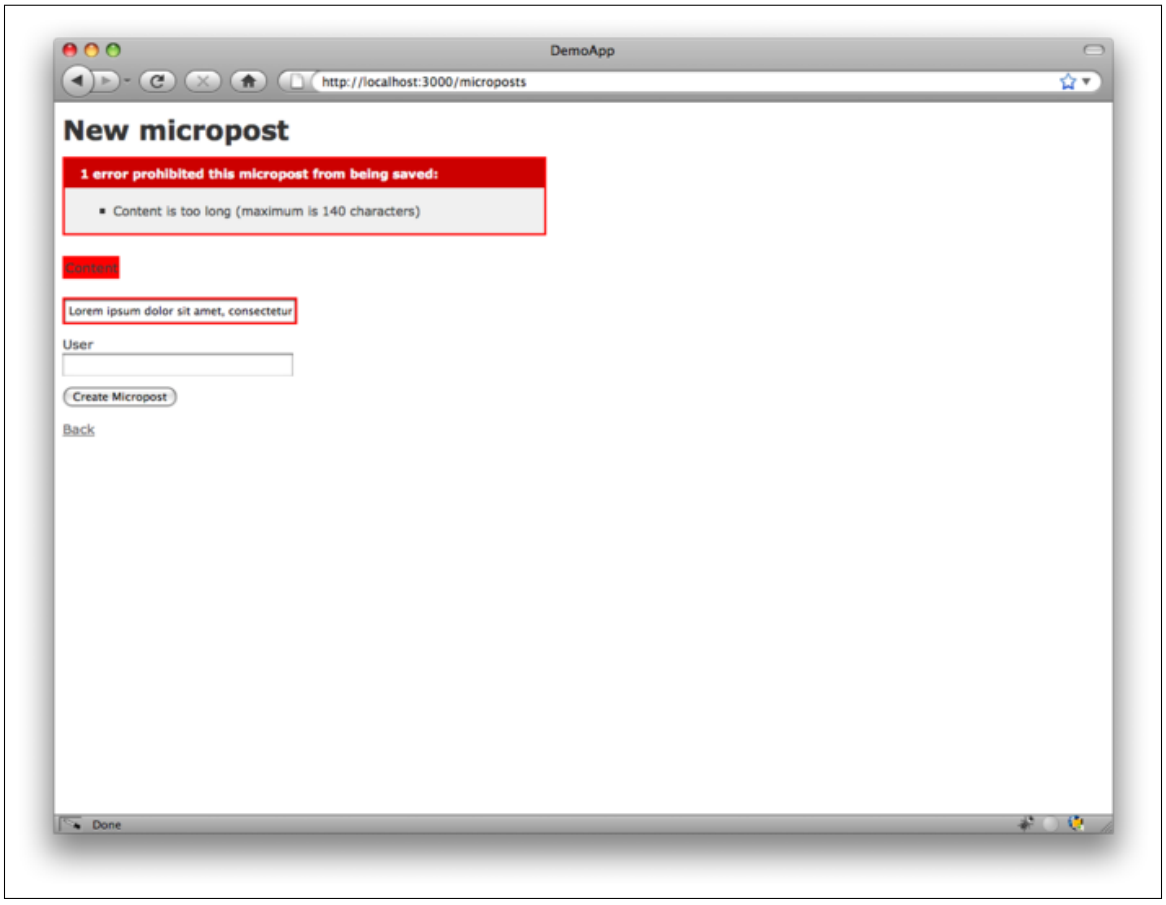

Figure 2.14: Error messages for a failed micropost creation. (full size)

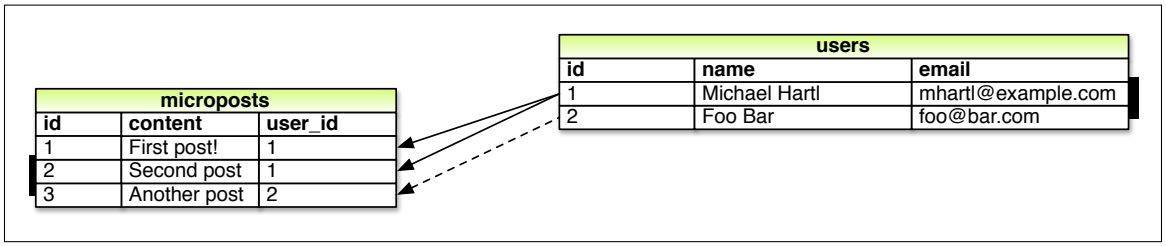

Figure 2.15: The association between microposts and users.

Listing 2.11. A micropost belongs to a user.
app/models/micropost.rb

```
class Micropost < ActiveRecord::Base
  attr_accessible :content, :user_id

  belongs_to :user

  validates :content, :length => { :maximum => 140 }
end
```

We can visualize the result of this association in Figure 2.15. Because of the **user_id** column in the **microposts** table, Rails (using Active Record) can infer the microposts associated with each user.

In Chapter 10 and Chapter 11, we will use the association of users and microposts both to display all a user's microposts and to construct a Twitter-like micropost feed. For now, we can examine the implications of the user-micropost association by using the *console*, which is a useful tool for interacting with Rails applications. We first invoke the console with **rails console** at the command line, and then retrieve the first user from the database using **User.first** (putting the results in the variable **first_user**):[7]

[7] Your console prompt might be something like **ruby-1.9.3-head >**, but the examples use >> since Ruby versions will vary.

```
$ rails console
>> first_user = User.first
=> #<User id: 1, name: "Michael Hartl", email: "michael@example.org",
created_at: "2011-11-03 02:01:31", updated_at: "2011-11-03 02:01:31">
>> first_user.microposts
=> [#<Micropost id: 1, content: "First micropost!", user_id: 1, created_at:
"2011-11-03 02:37:37", updated_at: "2011-11-03 02:37:37">, #<Micropost id: 2,
content: "Second micropost", user_id: 1, created_at: "2011-11-03 02:38:54",
updated_at: "2011-11-03 02:38:54">]
>> exit
```

(I include the last line just to demonstrate how to exit the console, and on most systems you can Ctrl-d for the same purpose.) Here we have accessed the user's microposts using the code `first_user.microposts`: with this code, Active Record automatically returns all the microposts with `user_id` equal to the id of `first_user` (in this case, `1`). We'll learn much more about the association facilities in Active Record in Chapter 10 and Chapter 11.

2.3.4 Inheritance hierarchies

We end our discussion of the demo application with a brief description of the controller and model class hierarchies in Rails. This discussion will only make much sense if you have some experience with object-oriented programming (OOP); if you haven't studied OOP, feel free to skip this section. In particular, if you are unfamiliar with *classes* (discussed in Section 4.4), I suggest looping back to this section at a later time.

We start with the inheritance structure for models. Comparing Listing 2.12 and Listing 2.13, we see that both the User model and the Micropost model inherit (via the left angle bracket `<`) from `ActiveRecord::Base`, which is the base class for models provided by ActiveRecord; a diagram summarizing this relationship appears in Figure 2.16. It is by inheriting from `ActiveRecord::Base` that our model objects gain the ability to communicate with the database, treat the database columns as Ruby attributes, and so on.

Listing 2.12. The `User` class, with inheritance.
`app/models/user.rb`

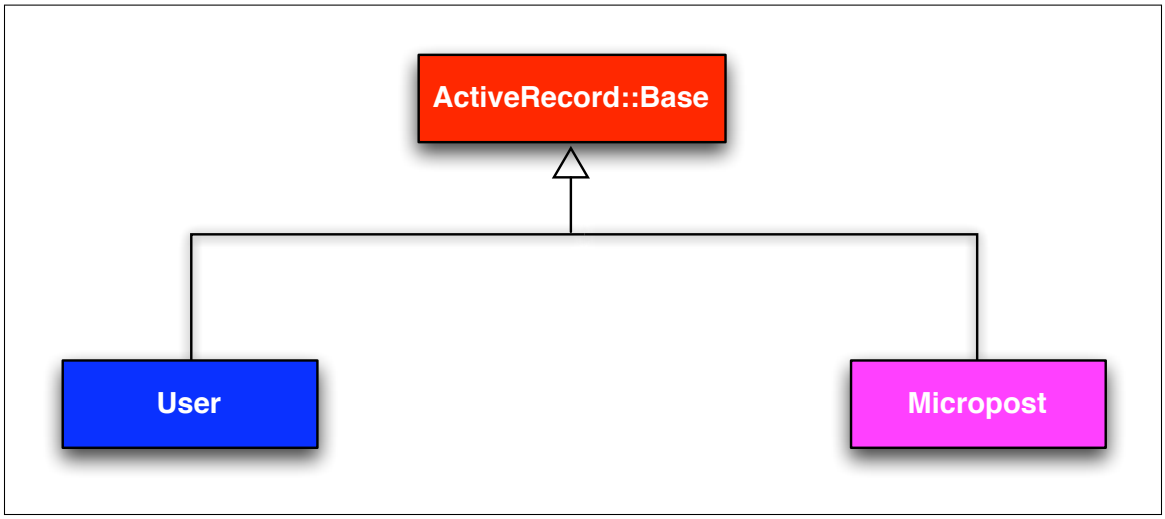

Figure 2.16: The inheritance hierarchy for the User and Micropost models.

```
class User < ActiveRecord::Base
  .
  .
  .
end
```

Listing 2.13. The `Micropost` class, with inheritance.
`app/models/micropost.rb`

```
class Micropost < ActiveRecord::Base
  .
  .
  .
end
```

The inheritance structure for controllers is only slightly more complicated. Comparing Listing 2.14 and Listing 2.15, we see that both the Users controller and the Microposts controller inherit from the Application controller. Examining Listing 2.16, we see that `ApplicationController` itself inherits from `ActionController::Base`; this is the base class for controllers provided

by the Rails library Action Pack. The relationships between these classes is illustrated in Figure 2.17.

Listing 2.14. The `UsersController` class, with inheritance.
`app/controllers/users_controller.rb`

```
class UsersController < ApplicationController
  .
  .
  .
end
```

Listing 2.15. The `MicropostsController` class, with inheritance.
`app/controllers/microposts_controller.rb`

```
class MicropostsController < ApplicationController
  .
  .
  .
end
```

Listing 2.16. The `ApplicationController` class, with inheritance.
`app/controllers/application_controller.rb`

```
class ApplicationController < ActionController::Base
  .
  .
  .
end
```

As with model inheritance, by inheriting ultimately from `ActionController::Base` both the Users and Microposts controllers gain a large amount of functionality, such as the ability to manipulate model objects, filter inbound HTTP requests, and render views as HTML. Since all Rails controllers inherit from `ApplicationController`, rules defined in the Application controller automatically apply to every action in the application. For example, in Section 8.2.1 we'll see how to include helpers for signing in and signing out of all of the sample application's controllers.

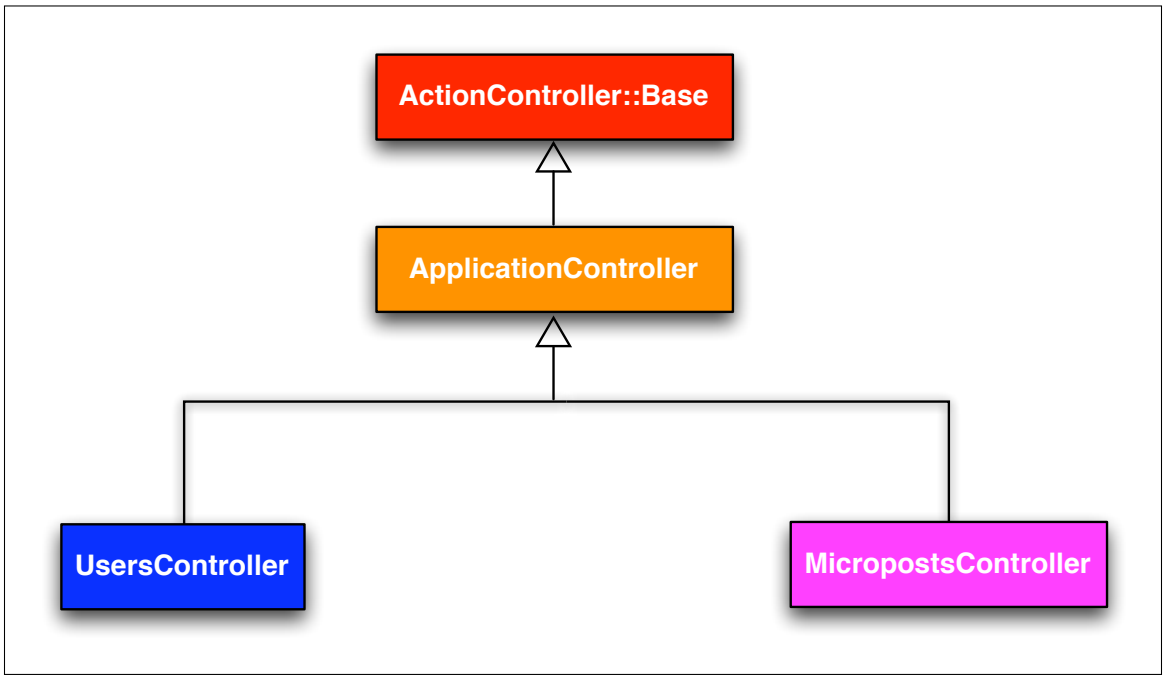

Figure 2.17: The inheritance hierarchy for the Users and Microposts controllers.

2.3.5 Deploying the demo app

With the completion of the Microposts resource, now is a good time to push the repository up to GitHub:

```
$ git add .
$ git commit -m "Finish demo app"
$ git push
```

Ordinarily, you should make smaller, more frequent commits, but for the purposes of this chapter a single big commit at the end is fine.

At this point, you can also deploy the demo app to Heroku as in Section 1.4:

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

Finally, migrate the production database (see below if you get a deprecation warning):

```
$ heroku run rake db:migrate
```

This updates the database at Heroku with the necessary user/micropost data model. You may get a deprecation warning regarding assets in `vendor/plugins`, which you should ignore since there aren't any plugins in that directory.

2.4 Conclusion

We've come now to the end of the 30,000-foot view of a Rails application. The demo app developed in this chapter has several strengths and a host of weaknesses.

Strengths

- High-level overview of Rails
- Introduction to MVC
- First taste of the REST architecture
- Beginning data modeling
- A live, database-backed web application in production

Weaknesses

- No custom layout or styling
- No static pages (like “Home” or “About”)
- No user passwords
- No user images
- No signing in
- No security
- No automatic user/micropost association
- No notion of “following” or “followed”
- No micropost feed
- No test-driven development
- **No real understanding**

The rest of this tutorial is dedicated to building on the strengths and eliminating the weaknesses.

Chapter 3

Mostly static pages

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

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

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

inscrutable code will suddenly look simple.

As in Chapter 2, before getting started we need to create a new Rails project, this time called **sample_app**:

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

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

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

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

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

gem 'rails', '3.2.11'

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

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

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

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

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

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

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

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

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

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

As in Section 1.4.1 and Chapter 2, we suppress the installation of production gems using the option `--without production`. This is a "remembered option", which means that we don't have to include it in future invocations of

[1] The successor to *Webrat*, Capybara is named after the world's largest rodent.

Bundler. Instead, we can write simply `bundle install` and production gems will be ignored automatically.[2]

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

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

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

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

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

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

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

```
# Ruby on Rails Tutorial: sample application

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

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

[2] In fact, you can even leave off `install`. The `bundle` command by itself is an alias for `bundle install`.

[3] As before, you may find the augmented file from Listing 1.7 to be more convenient depending on your system.

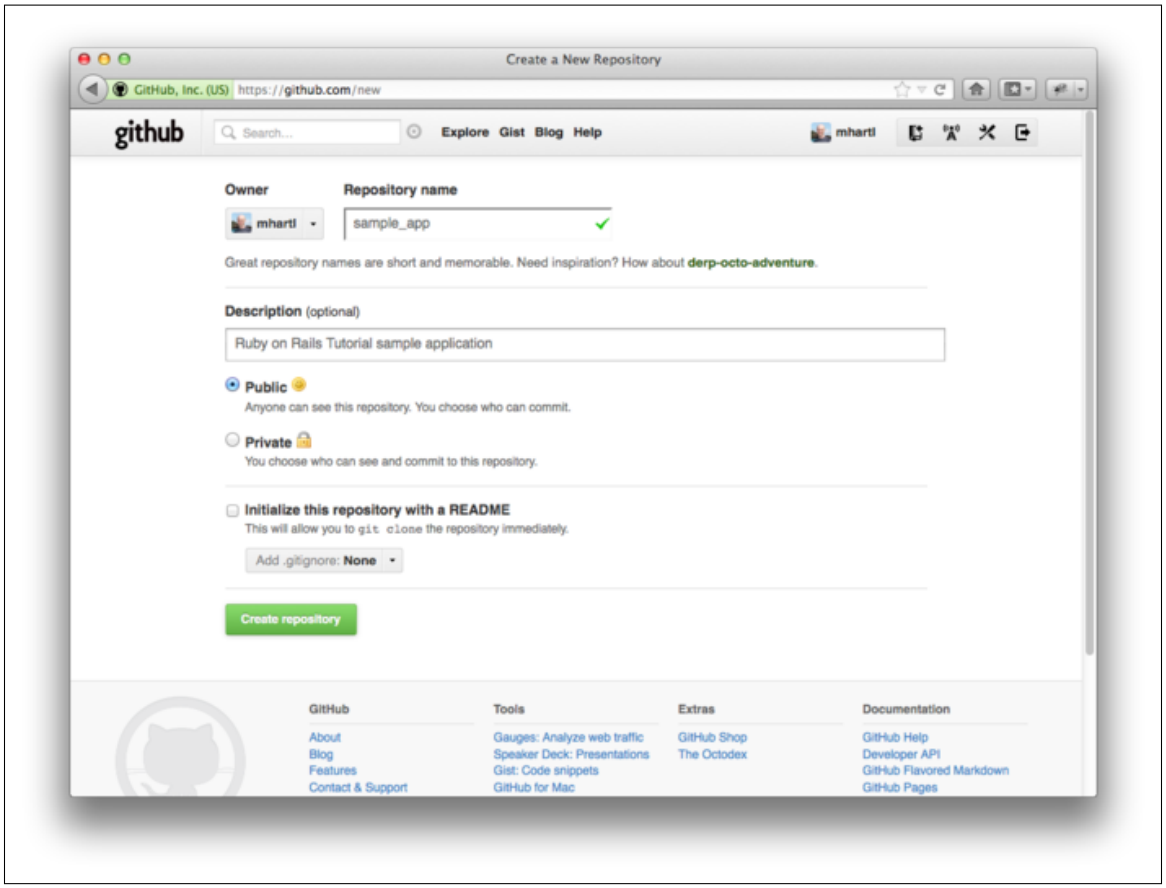

Figure 3.1: Creating the sample app repository at GitHub. (full size)

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

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

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

As a result of my performing this step, you can find the Rails Tutorial sample

application code on GitHub (under a slightly different name).[4]

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

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

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

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

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

```
$ heroku logs
```

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

3.1 Static pages

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

In order to get our bearings, it's helpful to recall the Rails directory structure from Section 1.2.3 (Figure 1.2). In this section, we'll be working mainly in

[4] https://github.com/railstutorial/sample_app_2nd_ed

the **app/controllers** and **app/views** directories. (In Section 3.2, we'll even add a new directory of our own.)

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

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

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

```
$ subl .
```

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

3.1.1 Truly static pages

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

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

[5] In fact, Rails ensures that requests for such files never hit the main Rails stack; they are delivered directly from the filesystem. (See *The Rails 3 Way* for more details.)

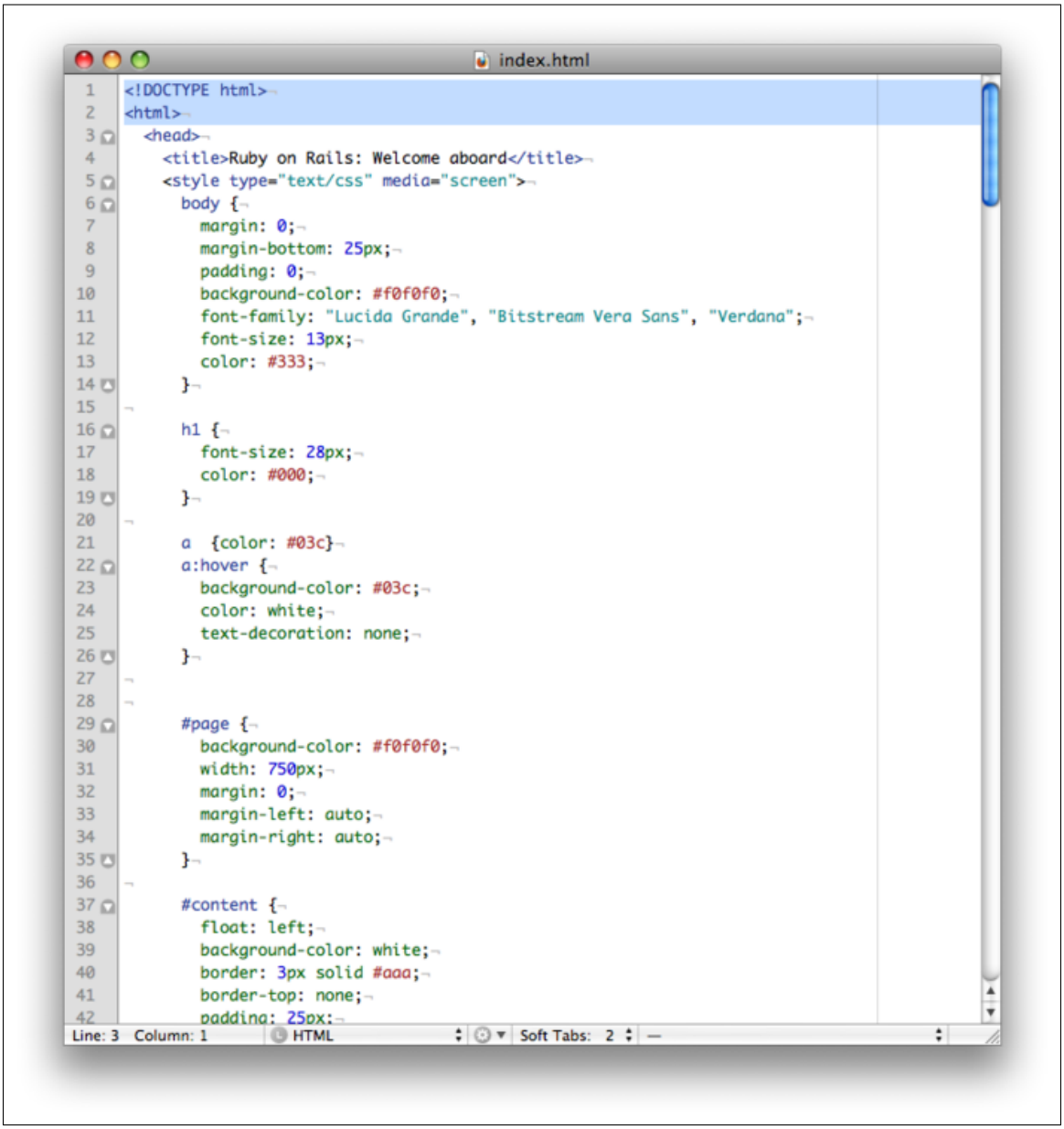

Figure 3.2: The **`public/index.html`** file. (full size)

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

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

Listing 3.3. A typical HTML file, with a friendly greeting.
`public/hello.html`

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

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

Now run a local server using

```
$ rails server
```

[6] As usual, replace `subl` with the command for your text editor.

[7] HTML changes with time; by explicitly making a doctype declaration we make it likelier that browsers will render our pages properly in the future. The extremely simple doctype `<!DOCTYPE html>` is characteristic of the latest HTML standard, HTML5.

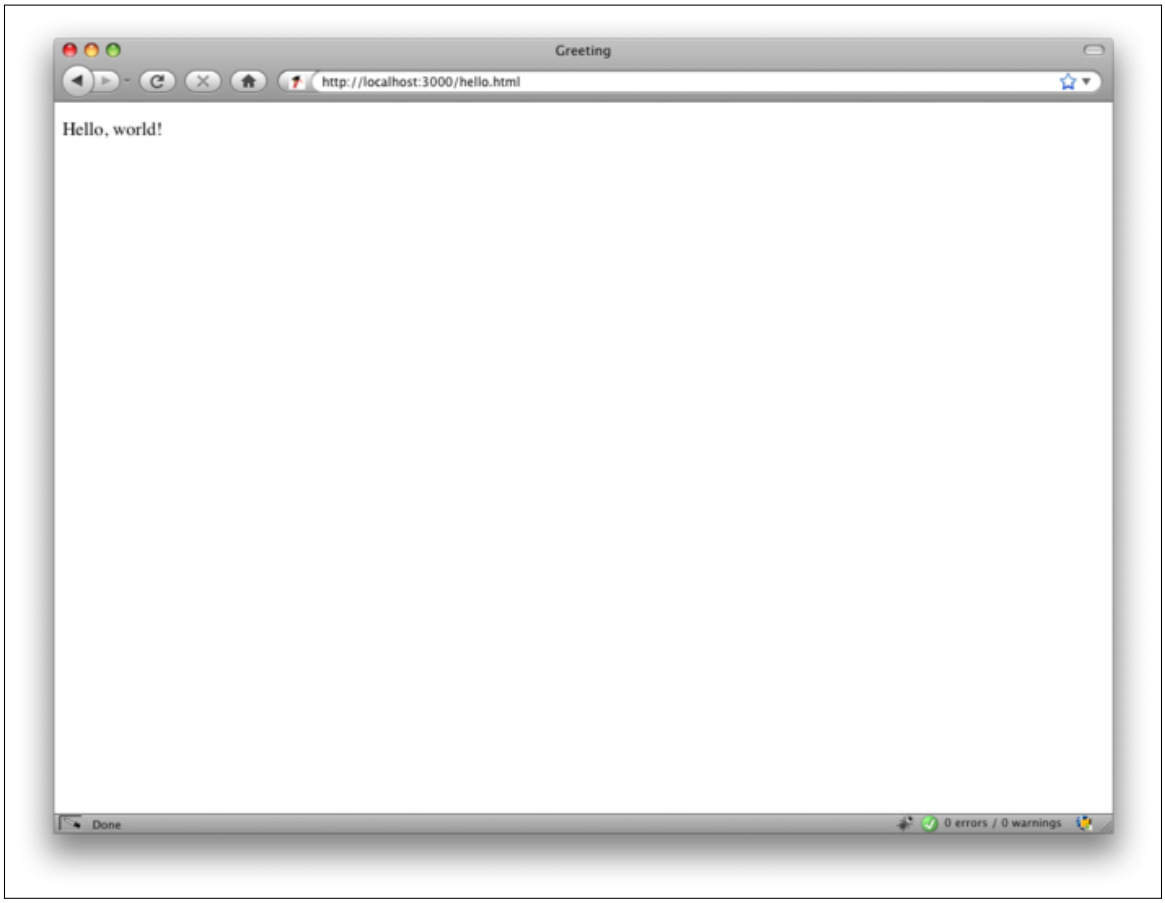

Figure 3.3: A new static HTML file. (full size)

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

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

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

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

3.1.2 Static pages with Rails

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

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

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

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

[8] Our method for making static pages is probably the simplest, but it's not the only way. The optimal method really depends on your needs; if you expect a *large* number of static pages, using a StaticPages controller can get quite cumbersome, but in our sample app we'll only need a few. See this blog post on simple pages at `has_many :through` for a survey of techniques for making static pages with Rails. *Warning:* The discussion is fairly advanced, so you might want to wait a while before trying to understand it.

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

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

Listing 3.4. Generating a StaticPages controller.

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

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

In Listing 3.4, note that we have passed the controller name as so-called CamelCase, which leads to the creation of a controller file written in snake case, so that a controller called StaticPages yields a file called **static_pages_controller.rb**. This is merely a convention, and in fact using snake case at the command line also works: the command

```
$ rails generate controller static_pages ...
```

also generates a controller called `static_pages_controller.rb`. Because Ruby uses CamelCase for class names (Section 4.4), my preference is to refer to controllers using their CamelCase names, but this is a matter of taste. (Since Ruby filenames typically use snake case, the Rails generator converts CamelCase to snake case using the `underscore` method.)

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

Box 3.1. Undoing things

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

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

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

Similarly, in Chapter 6 we'll generate a *model* as follows:

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

This can be undone using

```
$ rails destroy model Foo
```

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

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

```
$ rake db:migrate
```

We can undo a single migration step using

```
$ rake db:rollback
```

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

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

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

With these techniques in hand, we are well-equipped to recover from the inevitable development snafus.

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

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

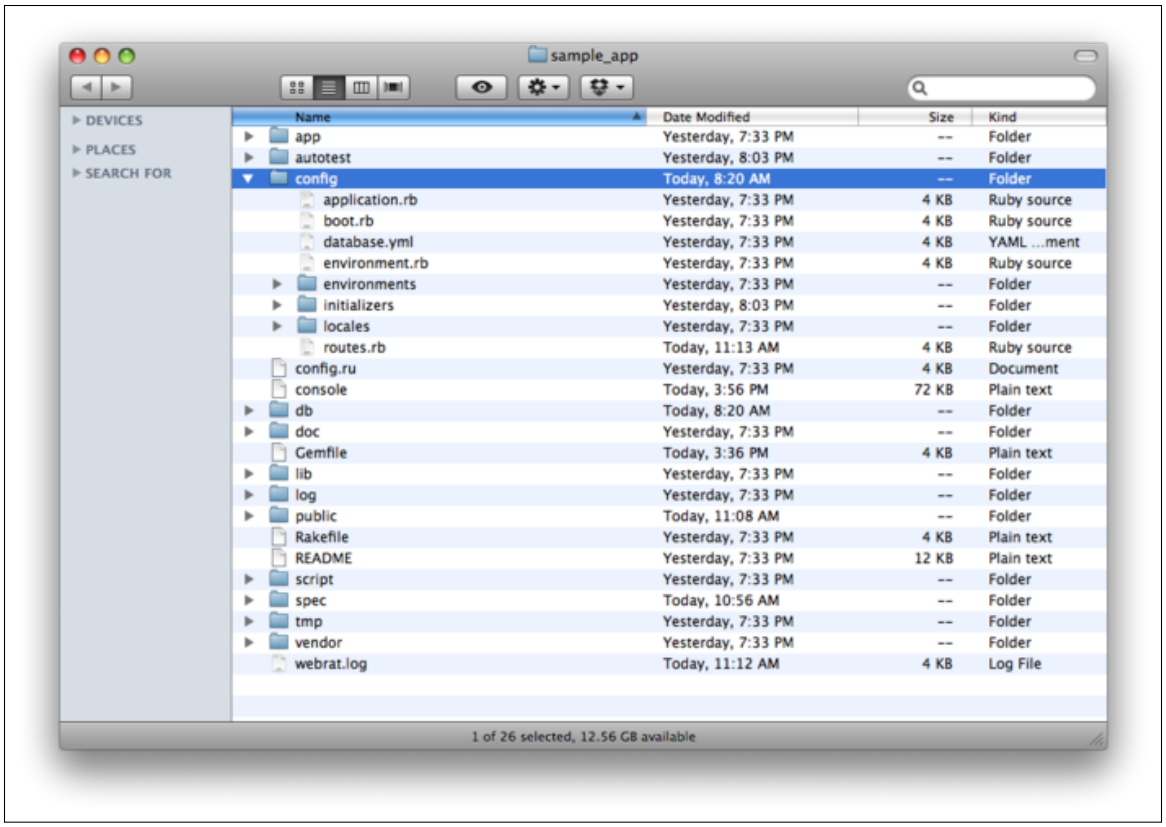

Figure 3.4: Contents of the sample app's `config` directory. (full size)

Listing 3.5. The routes for the `home` and `help` actions in the StaticPages controller.
config/routes.rb

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

Here the rule

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

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

Box 3.2. `GET`, et cet.

The hypertext transfer protocol (HTTP) defines four basic operations, corresponding to the four verbs *get*, *post*, *put*, and *delete*. These refer to operations between a *client* computer (typically running a web browser such as Firefox or Safari) and a *server* (typically running a web server such as Apache or Nginx). (It's important to understand that, when developing Rails applications on a local computer, the client and server are the same physical machine, but in general they are different.) An emphasis on HTTP verbs is typical of web

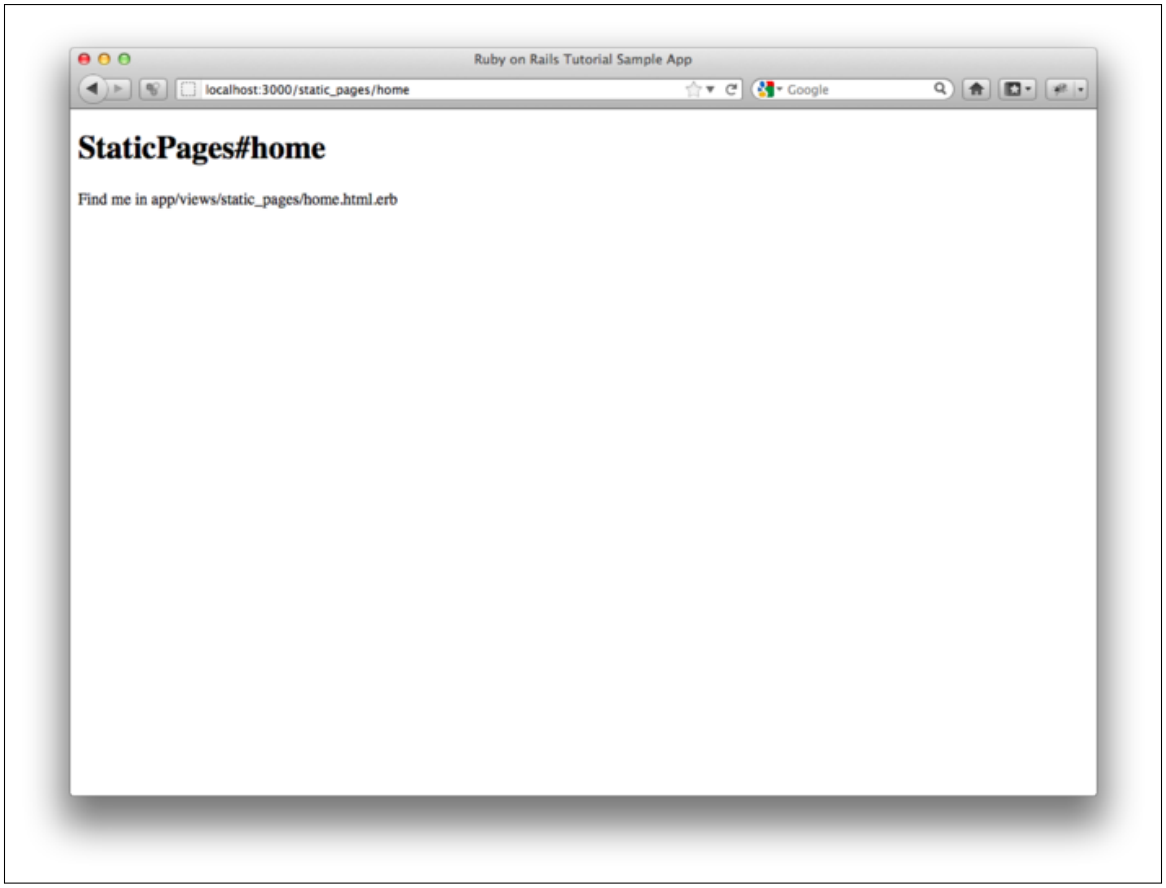

Figure 3.5: The raw home view (/static_pages/home). (full size)

frameworks (including Rails) influenced by the *REST architecture*, which we saw briefly in Chapter 2 and will start learning about more in Chapter 7.

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

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

Listing 3.6. The StaticPages controller made by Listing 3.4.
app/controllers/static_pages_controller.rb

```
class StaticPagesController < ApplicationController

  def home
  end

  def help
  end
end
```

We see from the **class** keyword in Listing 3.6 that **static_pages_controller.rb** defines a *class*, in this case called **StaticPagesCon-**

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

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

```
def home
end

def help
end
```

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

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

Listing 3.7. The generated view for the Home page.
app/views/static_pages/home.html.erb

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

The view for the `help` action is analogous (Listing 3.8).

Listing 3.8. The generated view for the Help page.
app/views/static_pages/help.html.erb

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

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

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

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

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

3.2 Our first tests

The *Rails Tutorial* takes an intuitive approach to testing that emphasizes the behavior of the application rather than its precise implementation, a variant of

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

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

It's important to understand that TDD is not always the right tool for the job: there's no reason to dogmatically insist that tests always should be written first, that they should cover every single feature, or that there should necessarily be any tests at all. For example, when you aren't at all sure how to solve a given programming problem, it's often useful to skip the tests and write only application code, just to get a sense of what the solution will look like. (In the language of Extreme Programming (XP), this exploratory step is called a *spike*.) Once you see the general shape of the solution, you can then use TDD to implement a more polished version.

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

3.2.1 Test-driven development

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

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

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

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

Listing 3.9. Code to test the contents of the Home page.
spec/requests/static_pages_spec.rb

```
require 'spec_helper'

describe "Static pages" do

  describe "Home page" do

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

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

Listing 3.9 contains a **describe** block with one *example*, i.e., a block starting with **it "..." do**:

```
describe "Home page" do

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

The first line indicates that we are describing the Home page. This is just a string, and it can be anything you want; RSpec doesn't care, but you and other human readers probably do. Then the spec says that when you visit the Home page at **/static_pages/home**, the content should contain the words "Sample App". As with the first line, what goes inside the quote marks is irrelevant to RSpec, and is intended to be descriptive to human readers. Then the line

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

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

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

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

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

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

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

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

Listing 3.10. Code to get a passing test for the Home page.
app/views/static_pages/home.html.erb

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

This arranges for a top-level heading (**<h1>**) with the content **Sample App**, which should get the test to pass. We also include an *anchor* tag **a**, which

[9] Running **bundle exec** every time is rather cumbersome; see Section 3.6 for some options to eliminate it.

[10] I actually use a dark background for both my terminal and editor, but the light background looks better in the screenshots.

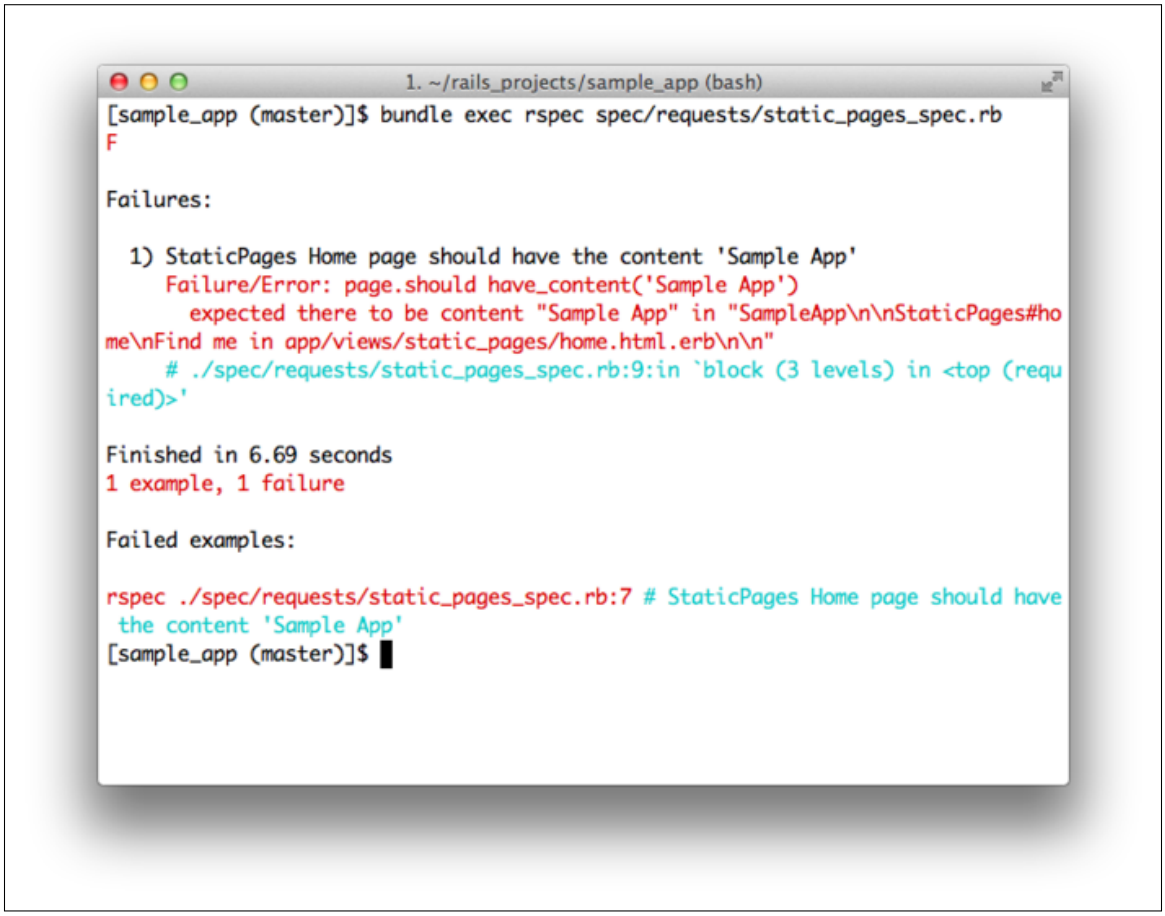

Figure 3.6: A red (failing) test. (full size)

creates links to the given URI (called an "href", or "hypertext reference", in the context of an anchor tag):

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

Now re-run the test to see the effect:

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

On my system, the passing test appears as in Figure 3.7.

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

Listing 3.11. Adding code to test the contents of the Help page.
spec/requests/static_pages_spec.rb

```
require 'spec_helper'

describe "Static pages" do

  describe "Home page" do

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

  describe "Help page" do

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

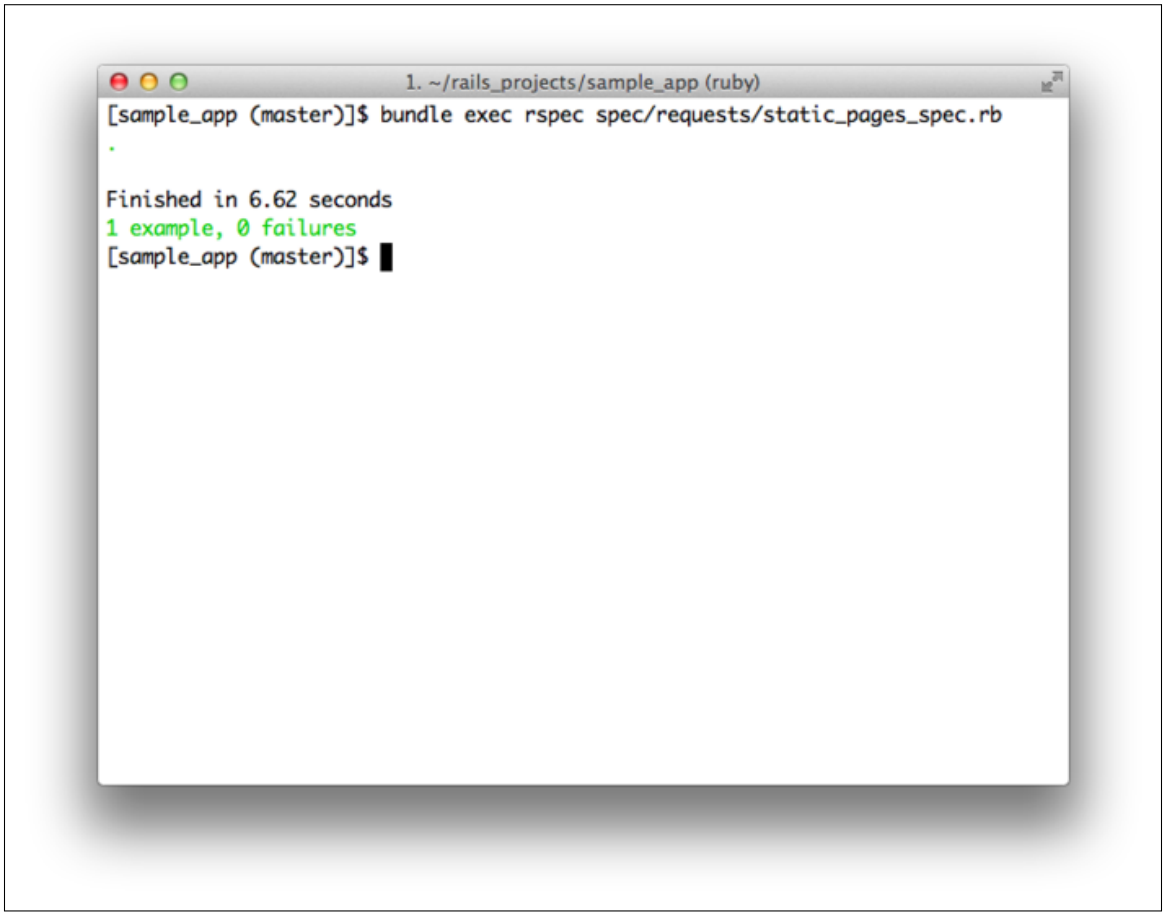

Figure 3.7: A green (passing) test. (full size)

Then run the tests:

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

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

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

Listing 3.12. Code to get a passing test for the Help page.
app/views/static_pages/help.html.erb

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

The tests should now pass:

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

3.2.2 Adding a page

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

Red

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

Listing 3.13. Adding code to test the contents of the About page.
spec/requests/static_pages_spec.rb

```
require 'spec_helper'

describe "Static pages" do

  describe "Home page" do

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

  describe "Help page" do

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

  describe "About page" do

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

Green

Recall from Section 3.1.2 that we can generate a static page in Rails by creating an action and corresponding view with the page's name. In our case, the About page will first need an action called **about** in the StaticPages controller. Hav-

ing written a failing test, we can now be confident that, in getting it to pass, we will actually have created a working About page.

If you run the RSpec example using

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

the output includes the following complaint:

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

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

Listing 3.14. Adding the `about` route.
`config/routes.rb`

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

Now running

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

complains that

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

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

Listing 3.15. The StaticPages controller with added **about** action.
app/controllers/static_pages_controller.rb

```
class StaticPagesController < ApplicationController

  def home
  end

  def help
  end

  def about
  end
end
```

Now running

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

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

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

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

Listing 3.16. Code for the About page.
app/views/static_pages/about.html.erb

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

Running RSpec should now get us back to Green:

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

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

Refactor

Now that we've gotten to Green, we are free to refactor our code with confidence. Oftentimes code will start to "smell", meaning that it gets ugly, bloated, or filled with repetition. The computer doesn't care, of course, but humans do, so it is important to keep the code base clean by refactoring frequently. Having a good test suite is an invaluable tool in this regard, as it dramatically lowers the probability of introducing bugs while refactoring.

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

3.3 Slightly dynamic pages

Now that we've created the actions and views for some static pages, we'll make them *very slightly* dynamic by adding some content that changes on a per-page

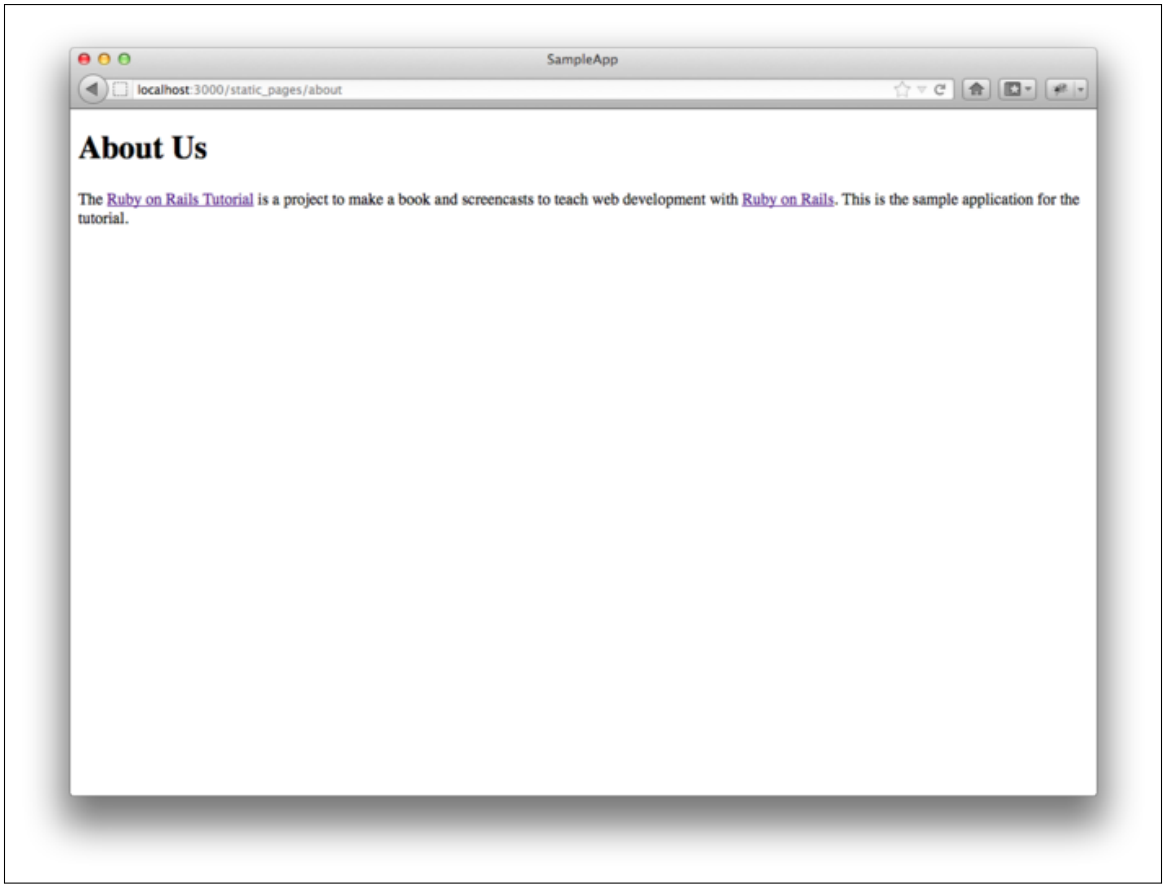

Figure 3.8: The new About page (/static_pages/about). (full size)

basis: we'll have the title of each page change to reflect its content. Whether a changing title represents *truly* dynamic content is debatable, but in any case it lays the necessary foundation for unambiguously dynamic content in Chapter 7.

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

3.3.1 Testing a title change

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

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

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

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

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

Listing 3.17. A title test.

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

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

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

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

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

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

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

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

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

will also match the full title.

Note that in Listing 3.17 we've broken the material inside **have_selector** into two lines; this tells you something important about Ruby syntax:

Ruby doesn't care about newlines.[11] The *reason* I chose to break the code into pieces is that I prefer to keep lines of source code under 80 characters for legibility.[12] As it stands, this code formatting is still rather ugly; Section 3.5 has a refactoring exercise that makes them prettier, and Section 5.3.4 completely rewrites the StaticPages tests to take advantage of the latest features in RSpec.

Adding new tests for each of our three static pages, following the model of Listing 3.17, gives us our new StaticPages test (Listing 3.18).

Listing 3.18. The StaticPages controller spec with title tests.
spec/requests/static_pages_spec.rb

```
require 'spec_helper'

describe "Static pages" do

  describe "Home page" do

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

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

  describe "Help page" do

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

    it "should have the title 'Help'" do
```

[11] A newline is what comes at the end of a line, thereby starting a new line. In code, it is represented by the character `\n`.

[12] Actually *counting* columns could drive you crazy, which is why many text editors have a visual aid to help you. For example, if you take a look back at Figure 1.1, you'll see a small vertical line on the right to help keep code under 80 characters. (It's actually at 78 columns, which gives you a little margin for error.) If you use TextMate, you can find this feature under `View > Wrap Column > 78`. In Sublime Text, you can use `View > Ruler > 78` or `View > Ruler > 80`.

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

  describe "About page" do

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

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

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

With the tests from Listing 3.18 in place, you should run

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

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

3.3.2 Passing title tests

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

Listing 3.19. The view for the Home page with full HTML structure.
`app/views/static_pages/home.html.erb`

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

Listing 3.19 uses the title tested for in Listing 3.18:

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

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

Listing 3.20. The view for the Help page with full HTML structure.
app/views/static_pages/help.html.erb

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

Listing 3.21. The view for the About page with full HTML structure.
app/views/static_pages/about.html.erb

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

3.3.3 Embedded Ruby

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

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

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

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

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

Listing 3.22. The view for the Home page with an Embedded Ruby title.
app/views/static_pages/home.html.erb

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

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

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

indicates using <% ... %> that Rails should call the **provide** function and associate the string **'Home'** with the label **:title**.[14] Then, in the title, we

[13] There is a second popular template system called Haml, which I personally love, but it's not *quite* standard enough yet for use in an introductory tutorial.

[14] Experienced Rails developers might have expected the use of **content_for** at this point, but it doesn't work well with the asset pipeline. The **provide** function is its replacement.

use the closely related notation `<%= ... %>` to insert the title into the template using Ruby's **`yield`** function:[15]

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

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

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

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

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

Listing 3.23. The view for the Help page with an Embedded Ruby title.
`app/views/static_pages/help.html.erb`

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

[15] If you've studied Ruby before, you might suspect that Rails is *yielding* the contents to a block, and your suspicion would be correct. But you don't need to know this to develop applications with Rails.

Listing 3.24. The view for the About page with an Embedded Ruby title.
app/views/static_pages/about.html.erb

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

3.3.4 Eliminating duplication with layouts

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

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

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

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

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

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

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

The resulting layout appears in Listing 3.25.

Listing 3.25. The sample application site layout.
app/views/layouts/application.html.erb

```
<!DOCTYPE html>
<html>
  <head>
    <title>Ruby on Rails Tutorial Sample App | <%= yield(:title) %></title>
    <%= stylesheet_link_tag    "application", :media => "all" %>
    <%= javascript_include_tag "application" %>
    <%= csrf_meta_tags %>
  </head>
  <body>
    <%= yield %>
  </body>
</html>
```

Note here the special line

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

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

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

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

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

Of course, the views in Listing 3.22, Listing 3.23, and Listing 3.24 are still filled with all the HTML structure included in the layout, so we have to remove it, leaving only the interior contents. The resulting cleaned-up views appear in Listing 3.26, Listing 3.27, and Listing 3.28.

Listing 3.26. The Home page with HTML structure removed.
`app/views/static_pages/home.html.erb`

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

Listing 3.27. The Help page with HTML structure removed.
`app/views/static_pages/help.html.erb`

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

Listing 3.28. The About page with HTML structure removed.
app/views/static_pages/about.html.erb

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

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

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

3.4 Conclusion

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

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

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

Then merge the changes back into the master branch using the same technique as in Section 1.3.5:

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

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

```
$ git push
```

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

```
$ git push heroku
```

3.5 Exercises

1. Make a Contact page for the sample app. Following the model in Listing 3.18, first write a test for the existence of a page at the URI /static_pages/contact by testing for the right `h1` content, and then write a second test for the title "Ruby on Rails Tutorial Sample App | Contact". Get your tests to pass, and then fill in the Contact page with the content from Listing 3.29. (This exercise is solved as part of Section 5.3.)

2. You may have noticed some repetition in the StaticPages controller spec (Listing 3.18). In particular, the base title, "Ruby on Rails Tutorial Sample App", is the same for every title test. Using the RSpec `let` function, which creates a variable corresponding to its argument, verify that the tests in Listing 3.30 still pass. Listing 3.30 introduces *string interpolation*, which is covered further in Section 4.2.2.

3. **(advanced)** As noted on the Heroku page on using `sqlite3` for development, it's a good idea to use the same database in development, test, and production environments to minimize the possibility of subtle incompatibilities. Follow the Heroku instructions for local PostgreSQL installation to install the PostgreSQL database on your local system. Update your **`Gemfile`** to eliminate the `sqlite3` gem and use the `pg` gem exclusively, as shown in Listing 3.31. You will also have to learn about the **`config/database.yml`** file and how to run PostgreSQL locally. Your goal should be to create and configure both the development database and the test database to use PostgreSQL. **Warning:** You may find this exercise challenging, and I recommend it only for advanced users. If you get stuck, don't hesitate to skip it; as noted previously, the sample application developed in this tutorial is fully compatible with both SQLite and PostgreSQL.

Listing 3.29. Code for a proposed Contact page.
`app/views/static_pages/contact.html.erb`

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

Listing 3.30. The StaticPages controller spec with a base title.
`spec/requests/static_pages_spec.rb`

```
require 'spec_helper'

describe "Static pages" do

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

  describe "Home page" do

    it "should have the h1 'Sample App'" do
```

```
      visit '/static_pages/home'
      page.should have_selector('h1', :text => 'Sample App')
    end

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

  describe "Help page" do

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

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

  describe "About page" do

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

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

  describe "Contact page" do

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

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

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

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

gem 'rails', '3.2.11'
gem 'pg', '0.12.2'

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

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

gem 'jquery-rails', '2.0.2'

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

3.6 Advanced setup

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

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

3.6.1 Eliminating `bundle exec`

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

RVM Bundler integration

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

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

rvm 1.15.6 (master)
```

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

```
$ rspec spec/
```

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

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

[16] http://rvm.io/integration/bundler/

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

Then run these:

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

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

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

```
# Ignore bundler config
/.bundle

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

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

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

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

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

binstubs

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

```
$ bundle --binstubs
```

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

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

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

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

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

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

3.6.2 Automated tests with Guard

One annoyance associated with using the **rspec** command is having to switch to the command line and run the tests by hand. (A second annoyance, the

slow start-up time of the test suite, is addressed in Section 3.6.3.) In this section, we'll show how to use Guard to automate the running of the tests. Guard monitors changes in the filesystem so that, for example, when we change the `static_pages_spec.rb` file only those tests get run. Even better, we can configure Guard so that when, say, the `home.html.erb` file is modified, the `static_pages_spec.rb` automatically runs.

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

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

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

gem 'rails', '3.2.11'

group :development, :test do
  gem 'sqlite3', '1.3.5'
  gem 'rspec-rails', '2.11.0'
  gem 'guard-rspec', '1.2.1'
end

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

gem 'jquery-rails', '2.0.2'

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

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

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

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

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

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

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

```
$ bundle install
```

Then initialize Guard so that it works with RSpec:

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

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

Listing 3.34. Additions to the default `Guardfile`.

```
require 'active_support/core_ext'

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

Here the line

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

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

We can now start `guard` as follows:

```
$ bundle exec guard
```

To eliminate the need to prefix the command with `bundle exec`, re-follow the steps in Section 3.6.1.

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

```
$ mkdir spec/routing
```

3.6.3 Speeding up tests with Spork

When running **bundle exec rspec**, you may have noticed that it takes several seconds just to start running the tests, but once they start running they finish quickly. This is because each time RSpec runs the tests it has to reload the entire Rails environment. The Spork test server[17] aims to solve this problem. Spork loads the environment *once*, and then maintains a pool of processes for running future tests. Spork is particularly useful when combined with Guard (Section 3.6.2).

The first step is to add the **spork** gem dependency to the **Gemfile** (Listing 3.35).

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

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

gem 'rails', '3.2.11'
.
.
.
group :development, :test do
  .
  .
  .
  gem 'guard-spork', '1.2.0'
  gem 'spork', '0.9.2'
end
.
.
.
```

Then install Spork using **bundle install**:

[17] A *spork* is a combination spoon-fork. The project's name is a pun on Spork's use of POSIX forks.

```
$ bundle install
```

Next, bootstrap the Spork configuration:

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

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

Listing 3.36. Adding environment loading to the **Spork.prefork** block.
spec/spec_helper.rb

```
require 'rubygems'
require 'spork'

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

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

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

    # Remove this line if you're not using ActiveRecord or ActiveRecord fixtures
```

```
    config.fixture_path = "#{::Rails.root}/spec/fixtures"

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

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

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

end
```

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

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

6 examples, 0 failures

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

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

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

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

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

6 examples, 0 failures

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

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

Listing 3.37. Configuring RSpec to automatically use Spork.
.rspec

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

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

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

Guard with Spork

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

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

We then need to change the **Guardfile** as in Listing 3.38.

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

```
require 'active_support/core_ext'

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

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

Note that we've updated the arguments to **guard** to include `:cli => --drb`, which ensures that Guard uses the command-line interface (cli) to the Spork server. We've also added a command to watch the **spec/support/** directory, which we'll start modifying in Chapter 5.

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

```
$ bundle exec guard
```

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

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

3.6.4 Tests inside Sublime Text

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

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

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

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

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

[18] http://railstutorial.org/screencasts

[19] https://github.com/maltize/sublime-text-2-ruby-tests

[20] https://github.com/mhartl/rails_tutorial_sublime_text

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

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

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