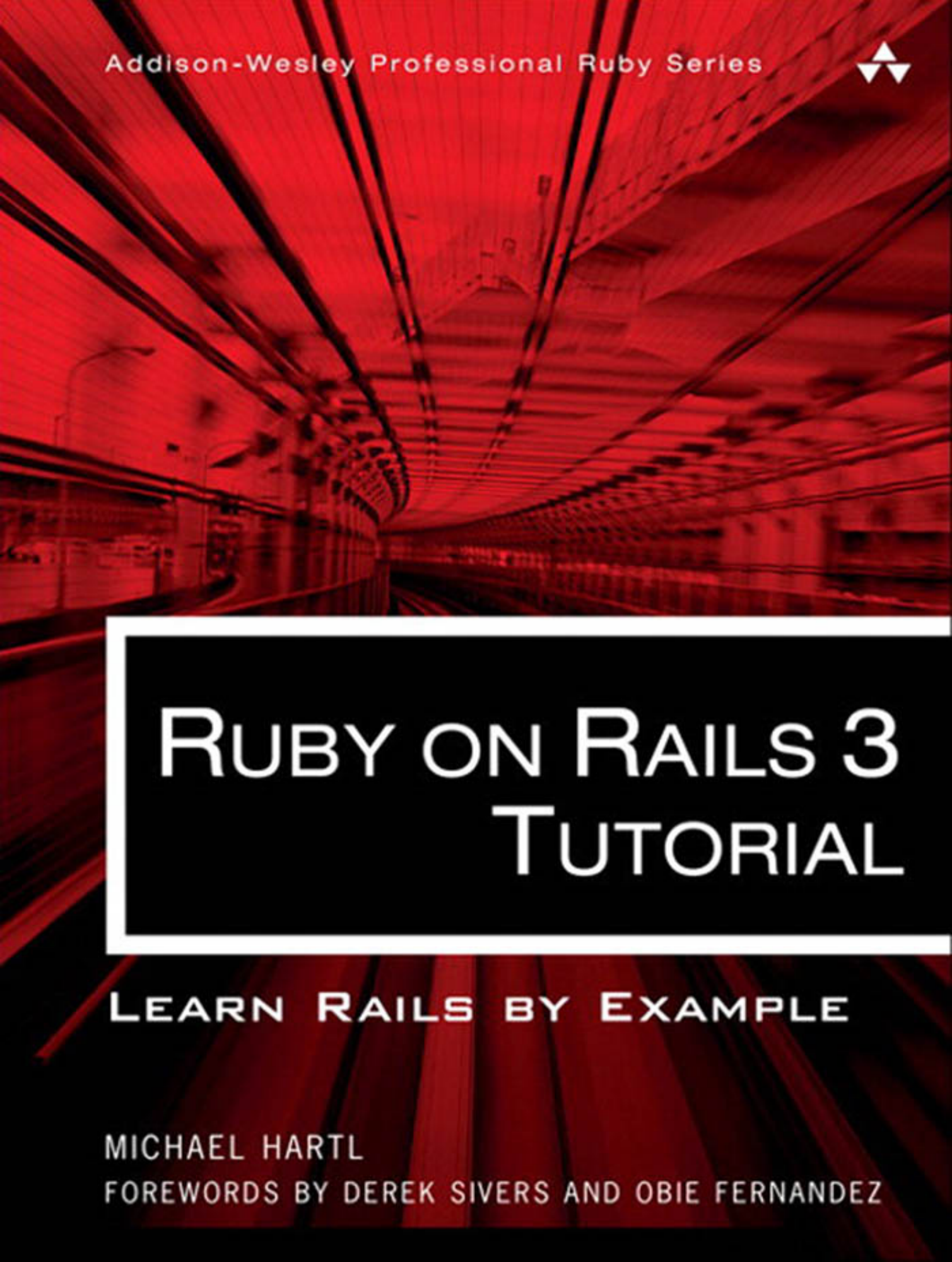
Addison-Wesley Professional Ruby Series
RUBY ON RAILS 3 TUTORIAL
LEARN RAILS BY EXAMPLE
MICHAEL HARTL
FOREWORDS BY DEREK SIVERS AND OBIE FERNANDEZ

Library of Congress Cataloging-in-Publication Data
Hartl, Michael.
Ruby on rails 3 tutorial : learn Rails by example / Michael Hartl.
p. cm.
Includes index.
ISBN-10: 0-321-74312-1 (pbk. : alk. paper)
ISBN-13: 978-0-321-74312-1 (pbk. : alk. paper)
1. Ruby on rails (Electronic resource) 2. Web site development. 3. Ruby (Computer program language) I. Title.
TK5105.8885.R83H37 2011
005.1′17–dc22 2010039450

ISBN 13: 978-0-321-74312-1
ISBN 10: 0-321-74312-1
Text printed in the United States on recycled paper at Edwards Brothers in Ann Arbor, Michigan
First printing, December 2010

Editor-in-Chief
Mark Taub

Executive Acquisitions Editor
Debra Williams Cauley

Managing Editor
John Fuller

Project Editor
Elizabeth Ryan

Copy Editor
Erica Orloff

Indexer
Claire Splan

Proofreader
Claire Splan

Publishing Coordinator
Kim Boedigheimer

Cover Designer
Gary Adair

Compositor
Glyph International

Contents

This page intentionally left blank

Foreword

My former company (CD Baby) was one of the first to loudly switch to Ruby on Rails, and then even more loudly switch back to PHP (Google me to read about the drama). This book by Michael Hartl came so highly recommended that I had to try it, and *Ruby on Rails™ 3 Tutorial* is what I used to switch back to Rails again.

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

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

Enjoy!

—Derek Sivers (sivers.org)
Founder, CD Baby and Thoughts, Ltd.

This page intentionally left blank

Foreword

"If I want to learn web development with Ruby on Rails, how should I start?" For years Michael Hartl has provided the answer as author of the *RailsSpace* tutorial in our series and now the new *Ruby on Rails™ 3 Tutorial* that you hold in your hands (or PDF reader, I guess.)

I'm so proud of having Michael on the series roster. He is living, breathing proof that we Rails folks are some of the luckiest in the wide world of technology. Before getting into Ruby, Michael taught theoretical and computational physics at Caltech for six years, where he received the Lifetime Achievement Award for Excellence in Teaching in 2000. He is a Harvard graduate, has a Ph.D. in Physics from Caltech, and is an alumnus of Paul Graham's esteemed Y Combinator program for entrepreneurs. And what does Michael apply his impressive experience and teaching prowess to? Teaching new software developers all around the world how to use Ruby on Rails effectively! Lucky we are indeed!

The availability of this tutorial actually comes at a critical time for Rails adoption. We're five years into the history of Rails and today's version of the platform has unprecedented power and flexibility. Experienced Rails folks can leverage that power effectively, but we're hearing growing cries of frustration from newcomers. The amount of information out there about Rails is fantastic if you know what you're doing already. However, if you're new, the scope and mass of information about Rails can be mind-boggling.

Luckily, Michael takes the same approach as he did in his first book in the series, building a sample application from scratch, and writes in a style that's meant to be read from start to finish. Along the way, he explains all the little details that are likely to trip up beginners. Impressively, he goes beyond just a straightforward explanation of what Rails does and ventures into prescriptive advice about good software development

practices, such as test-driven development. Neither does Michael constrain himself to a box delineated by the extents of the Rails framework—he goes ahead and teaches the reader to use tools essential to existence in the Rails community, such as Git and GitHub. In a friendly style, he even provides copious contextual footnotes of benefit to new programmers, such as the pronunciation of SQL and pointers to the origins of *lorem ipsum*. Tying all the content together in a way that remains concise and usable is truly a tour de force of dedication!

I tell you with all my heart that this book is one of the most significant titles in my Professional Ruby Series, because it facilitates the continued growth of the Rails ecosystem. By helping newcomers become productive members of the community quickly, he ensures that Ruby on Rails continues its powerful and disruptive charge into the mainstream. The Rails Tutorial is potent fuel for the fire that is powering growth and riches for so many of us, and for that we are forever grateful.

—Obie Fernandez, Series Editor

This page intentionally left blank

CHAPTER 1
From Zero to Deploy

Welcome to *Ruby on Rails™ 3 Tutorial: Learn Rails by Example*. 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 *Ruby on Rails Tutorial*, you will have all the knowledge you need to develop and deploy your own custom web applications. 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 *Ruby on Rails Tutorial* uses Rails 3.0, the knowledge you gain here will be fully up to date with the latest and greatest version of Rails.[1]

Ruby on Rails Tutorial follows essentially the same approach as my previous Rails book,[2] teaching web development with Rails 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 *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 setting up our development environment (Section 1.2). We'll then create our first Rails application, called (appropriately enough) `first_app`. *Rails Tutorial* emphasizes good software development practices, so immediately after creating our fresh

1. The most up-to-date version of *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. In addition, PDF books purchased through railstutorial.org will continue to be updated as long as Rails 3.0 and RSpec 2.0 are still under active development.

2. *RailsSpace*, by Michael Hartl and Aurelius Prochazka (Addison-Wesley, 2007).

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 will be 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 *URLs*[3] using a web browser.

In Chapter 3, we'll create a *sample application* (called `sample_app`), this time 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 10, 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 11 and Chapter 12 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, is also written in Rails. Though of necessity our efforts will focus on this specific sample application, the emphasis throughout *Rails Tutorial* will be on general principles, so that you will have a solid foundation no matter 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, now updated as the 15-minute weblog using Rails 2 by Ryan Bates. These videos 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;

3. *URL* stands for Uniform Resource Locator. In practice, it is usually equivalent to "the thing you see in the address bar of your browser". By the way, the current preferred term is *URI*, for Uniform Resource Identifier, but popular usage still tilts toward *URL*.

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 *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 *Rails Tutorial* is the sample app, which we'll start writing in Chapter 3. At each stage of developing the sample application, we will generate *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. Rails users run the gamut from scrappy startups to huge companies: Posterous, UserVoice, 37signals, Shopify, Scribd, Twitter, 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 percent open-source, available under the permissive MIT License, and as a result it also costs nothing to download and 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 URLs—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. (Anyone who has attended a talk by Merb developer and Rails core team member Yehuda Katz can't help but notice what an *extremely* good idea it was to bring the Merb team on board.)

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

Rails Tutorial contains integrated tutorials not only for Rails, but also for the underlying Ruby language, as well as for HTML, CSS, some 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.

Rails derives much of its power from "magic"—that is, framework features (such as automatically inferring object attributes from database columns) that accomplish miracles but whose mechanisms can be rather mysterious. *Ruby on Rails Tutorial* is *not* designed to explain this magic—mainly because most Rails application developers never need to know what's behind the curtain. (After all, Ruby itself is mostly written in the C programming language, but you don't have to dig into the C source to use Ruby.) If you're a confirmed pull-back-the-curtain kind of person, I recommend *The Rails 3 Way* by Obie Fernandez as a companion volume to *Ruby on Rails Tutorial*.

Although this book has no formal prerequisites, you should of course have at least *some* computer experience. If you've never even used a text editor before, it will be tough going, but with enough determination you can probably soldier through. If, on the other hand, your `.emacs` file is so complex it could make a grown man cry, there is still plenty of material to keep you challenged. *Rails Tutorial* is designed to teach Rails development no matter what your background is, but your path and reading experience will depend on your particular circumstances.

All readers: One common question when learning Rails is whether to learn Ruby first. The answer depends on your personal learning style. If you prefer to learn everything systematically from the ground up, then learning Ruby first might work well for you, and there are several book recommendations in this section to get you started. 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. Moreover, the subset of Ruby needed by Rails developers is different from what you'll find in a pure-Ruby introduction, whereas *Rails Tutorial* focuses on exactly that subset. If your primary interest is making web applications, I recommend starting with *Rails Tutorial* and then reading a book on pure Ruby next. It's not an all-or-nothing proposition, though: if you start reading *Rails Tutorial* and feel your (lack of) Ruby knowledge holding you back, feel free to switch to a Ruby book and come back when you feel ready. You might also consider getting a taste of Ruby by following a short online tutorial, such as can be found at http://www.ruby-lang.org/ or http://rubylearning.com/.

Another common question is whether to use tests from the start. As noted in the introduction, *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, feel free to skip them on first reading.[4] Indeed, some readers may find the inclusion of so many moving parts—such as tests, version control, and deployment—a bit overwhelming at first, and if you find yourself expending excessive energy on any of these steps, *don't hesitate to skip them.* Although I have included only material I consider essential to developing professional-grade Rails applications, only the core application code is strictly necessary the first time through.

Inexperienced programmers (non-designers): *Rails Tutorial* doesn't assume any background other than general computer knowledge, so if you have limited programming experience this book is a good place to start. Please bear in mind that it is only the first step on a long journey; web development has many moving parts, including HTML/CSS, JavaScript, databases (including SQL), version control, and deployment. This book contains short introductions to these subjects, but there is much more to learn.

Inexperienced programmers (designers): Your design skills give you a big leg up, since you probably already know HTML and CSS. After finishing this book you will be in an excellent position to work with existing Rails projects and possibly start some of your own. You may find the programming material challenging, but the Ruby language is unusually friendly to beginners, especially those with an artistic bent.

4. In practice, this will involve omitting all files with `spec` in their name, as we will start to see in Section 3.2.2.

After finishing *Ruby on Rails Tutorial*, I recommend that newer programmers read *Beginning Ruby* by Peter Cooper, which shares the same basic instructional philosophy as *Rails Tutorial*. I also recommend *The Ruby Way* by Hal Fulton. Finally, to gain a deeper understanding of Rails, I recommend *The Rails 3 Way* by Obie Fernandez.

Web applications, even relatively simple ones, are by their nature fairly complex. If you are completely new to web programming and find *Rails Tutorial* overwhelming, it could be that you're not quite ready to make web applications yet. In that case, I'd suggest learning the basics of HTML and CSS and then giving *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*, which starts with sample applications much smaller than a full-blown web app.

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.) *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 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 *Ruby on Rails Tutorial*, I recommend that experienced (non-Ruby) 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 will be ready for the more intermediate-to-advanced Rails resources. Here are some I particularly recommend:

- Railscasts: Excellent free Rails screencasts.
- PeepCode, Pragmatic.tv, EnvyCasts: Excellent commercial screencasters.
- Rails Guides: Good topical and up-to-date Rails references. *Rails Tutorial* refers frequently to the *Rails Guides* for more in-depth treatment of specific topics.
- Rails blogs: Too many to list, but there are tons of good ones.

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. First, 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]

Second, your humble author is a Linux/OS X kind of guy, and hasn't used Windows as his primary OS for more than a decade; as a result, *Rails Tutorial* has an unmistakable Unix flavor.[6] For example, in this book all command line examples use a Unix-style command line prompt (a dollar sign):

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

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
```

Rails Tutorial will also use Unix-style forward slashes as directory separators; my Rails Tutorial sample app, for instance, lives in

```
/Users/mhartl/rails_projects/first_app
```

The root directory for any given app is known as the *Rails root*, and henceforth all directories will be relative to this directory. For example, the **`config`** directory of my sample application is in

```
/Users/mhartl/rails_projects/first_app/config
```

This means that when referring to the file

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

I'll omit the Rails root and write **`config/routes.rb`** for brevity.

5. When reading *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`**.

6. Indeed, the entire Rails community has this flavor. In a full room at RailsConf you'll see a handful of PCs in a sea of MacBooks—with probably half the PCs running Linux. You can certainly develop Rails apps on Microsoft Windows, but you'll definitely be in the minority.

Finally, *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. In addition, 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.

1.2 Up and Running

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 themes: text editor/command line environments, and integrated development environments (IDEs). Let's consider the latter first.

IDEs

There is no shortage of Rails IDEs; indeed, the main Ruby on Rails site names four: RadRails, RubyMine, 3rd Rail, and NetBeans. All are cross-platform, and I've heard good things about several of them. I encourage you to try them and see if they work for you, but I have a confession to make: I have never found an IDE that met all my Rails development needs—and for some projects I haven't even been able to get them to work at all.

Text Editors and Command Lines

What are we to use to develop Rails apps, if not some awesome all-in-one IDE? I'd guess the majority of Rails developers opt for the same solution I've chosen: 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:

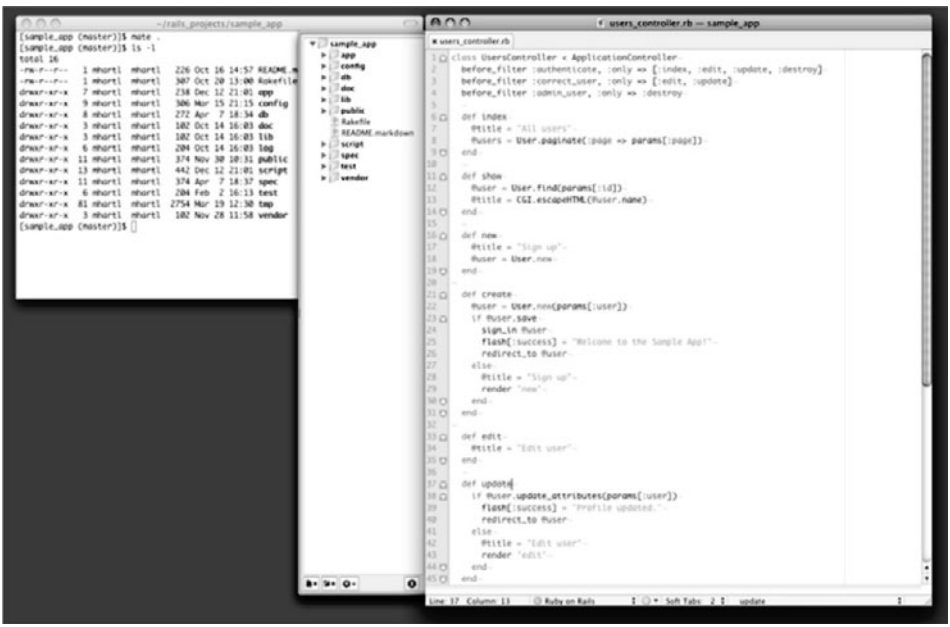

Figure 1.1 A text editor/command line development environment (TextMate/iTerm).

- **Macintosh OS X:** Like many Rails developers, I prefer TextMate. Other options include Emacs and MacVim (launched with the command **`mvim`**), the excellent Macintosh version of Vim.[7] I use iTerm for my command line terminal; others prefer the native Terminal app.
- **Linux:** Your editor options are basically the same as OS X, minus TextMate. I'd recommend graphical Vim (gVim), gedit (with the GMate plugins), or Kate. As far as command lines go, you're totally set: every Linux distribution comes with at least one command line terminal application (and often several).
- **Windows:** Unfortunately, I can't make any personal recommendations here, but you can do what I did: drop "rails windows" into Google to see what the latest thinking is on setting up a Rails development environment on Windows. Two combinations look especially promising: Vim for Windows with Console (recommended by Akita On Rails) or the E Text Editor with Console and Cygwin (recommended by Ben

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

Kittrell). Rails Tutorial readers have suggested looking at Komodo Edit (cross-platform) and the Sublime Text editor (Windows only) as well. No matter which editor you choose, I recommend trying Cygwin, which provides the equivalent of a Unix terminal under Windows; see, for example, this video on Ruby on Rails + Cygwin + Windows Vista. (In addition to installing the packages in the video, I recommend installing `git`, `curl`, and `vim`. Don't install Rails as in the video, though; use the instructions below instead.) With Cygwin, most of the command-line examples in the book should work with minimum modification.

If you go with some flavor of Vim, be sure to tap into the thriving community of Vim-using Rails hackers. See especially the rails.vim enhancements and the NERD tree project drawer.

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, Firebug Lite works with most other browsers, and both Safari and Chrome have a built-in "Inspect element" feature available by right-clicking on any part of the page. Regardless of which browser you use, experience shows that the time spent learning such a web inspector tool will be richly rewarded.

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 TextMate 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; in the end the reward is worth the effort.

1.2.2 Ruby, RubyGems, Rails, and Git

Now it's time to install Ruby and Rails. The canonical up-to-date source for this step is the Ruby on Rails download page. I'll assume you can go there now; parts of this book

can be read profitably offline, but not this part. I'll just inject some of my own comments on the steps.

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 1.9.2
```

to see the version number. Rails 3 requires Ruby 1.8.7 or later and works best with Ruby 1.9.2. This tutorial assumes that you are using the latest development version of Ruby 1.9.2, known as Ruby 1.9.2-head, but Ruby 1.8.7 should work as well.

The Ruby 1.9 branch is under heavy development, so unfortunately installing the latest Ruby can be quite a challenge. You will likely have to rely on the web for the most up-to-date instructions. What follows is a series of steps that I've gotten to work on my system (Macintosh OS X), but you may have to search around for steps that work on your system.

As part of installing Ruby, if you are using OS X or Linux I strongly recommend installing Ruby 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 Rails 3 and Rails 2.3 on the same machine. If you want to go this route, I suggest using RVM to install two Ruby/Rails combinations: Ruby 1.8.7/Rails 2.3.10 and Ruby 1.9.2/Rails 3.0.1. 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).[8]

8. 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.

After installing RVM, you can install Ruby as follows:[9]

```
$ rvm update --head
$ rvm reload
$ rvm install 1.8.7
$ rvm install 1.9.2
<wait a while>
```

Here the first two commands update and reload RVM itself, which is a good practice since RVM gets updated frequently. The final two commands do the actual Ruby installations; depending on your system, they might take a while to download and compile, so don't worry if it seems to be taking forever. (Also beware that lots of little things can go wrong. For example, on my system the latest version of Ruby 1.8.7 won't compile; instead, after much searching and hand-wringing, I discovered that I needed "patchlevel" number 174:

```
$ rvm install 1.8.7-p174
```

When things like this happen to you, it's always frustrating, but at least you know that it happens to everyone. . .)

Ruby programs are typically distributed via *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. In particular, Rails is distributed as a gem, and there are conflicts between Rails 2 and Rails 3, so if you want to run multiple versions of Rails on the same system you need to create a separate gemset for each:

```
$ rvm --create 1.8.7-p174@rails2tutorial
$ rvm --create use 1.9.2@rails3tutorial
```

Here the first command creates the gemset **`rails2tutorial`** associated with Ruby 1.8.7-p174, while the second command creates the gemset **`rails3tutorial`**

9. You might have to install the Subversion version control system to get this to work.

associated with Ruby 1.9.2 and uses it (via the **use** command) at the same time. RVM supports a large variety of commands for manipulating gemsets; see the documentation at http://rvm.beginrescueend.com/gemsets/.

In this tutorial, we want our system to use Ruby 1.9.2 and Rails 3.0 by default, which we can arrange as follows:

```
$ rvm --default use 1.9.2@rails3tutorial
```

This simultaneously sets the default Ruby to 1.9.2 and the default gemset to **rails3-tutorial**.

By the way, 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 tons of great 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-head/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:

```
$ [sudo] ruby setup.rb
```

Here **sudo** executes the command **ruby setup.rb** as an administrative user, which has access to files and directories that normal users can't touch; I have put it in brackets to indicate that using **sudo** may or may not be necessary for your particular system. Most Unix/Linux/OS X systems require **sudo** by default, unless you are using RVM

as suggested in Section 1.2.2. Note that you should *not* actually type any brackets; you should run either

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

or

```
$ ruby setup.rb
```

depending on your system.

If you already have RubyGems installed, you might want to update your system to the latest version:

```
$ [sudo] gem update --system
```

Finally, if you're using Ubuntu Linux, you might want to take a look at the Ubuntu/Rails 3.0 blog post by Toran Billups for full installation instructions.

Install Rails

Once you've installed RubyGems, installing Rails 3.0 should be easy:

```
$ [sudo] gem install rails --version 3.0.1
```

To verify that this worked, run the following command:

```
$ rails -v
Rails 3.0.1
```

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 1.1 Running the **`rails`** script to generate a new application.

```
$ mkdir rails_projects
$ cd rails_projects
$ rails new first_app
      create
      create  README
      create  .gitignore
      create  Rakefile
      create  config.ru
      create  Gemfile
      create  app
      create  app/controllers/application_controller.rb
      create  app/helpers/application_helper.rb
      create  app/views/layouts/application.html.erb
      create  app/models
      create  config
      create  config/routes.rb
      create  config/application.rb
      create  config/environment.rb
      .
      .
      .
```

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.

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. This involves opening the **`Gemfile`** with your favorite text editor:

```
$ cd first_app/
$ mate Gemfile
```

The result should look something like Listing 1.2.

Figure 1.2 The directory structure for a newly hatched Rails app.

Listing 1.2 The default **`Gemfile`** in the **`first_app`** directory.

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

gem 'rails', '3.0.1'

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

gem 'sqlite3-ruby', :require => 'sqlite3'

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

```
# Deploy with Capistrano
# gem 'capistrano'

# To use debugger
# gem 'ruby-debug'

# Bundle the extra gems:
# gem 'bj'
# gem 'nokogiri', '1.4.1'
# gem 'sqlite3-ruby', :require => 'sqlite3'
# gem 'aws-s3', :require => 'aws/s3'

# Bundle gems for certain environments:
# gem 'rspec', :group => :test
# group :test do
#   gem 'webrat'
# end
```

Table 1.1 A summary of the default Rails directory structure

File/Directory	Purpose
`app/`	Core application (app) code, including models, views, controllers, and helpers
`config/`	Application configuration
`db/`	Files to manipulate the database
`doc/`	Documentation for the application
`lib/`	Library modules
`log/`	Application log files
`public/`	Data accessible to the public (e.g., web browsers), such as images and cascading style sheets (CSS)
`script/rails`	A script provided by Rails for generating code, opening console sessions, or starting a local web 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
`README`	A brief description of the application
`Rakefile`	Utility tasks available via the **`rake`** command
`Gemfile`	Gem requirements for this app
`config.ru`	A configuration file for Rack middleware
`.gitignore`	Patterns for files that should be ignored by Git

Most 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, 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. Unfortunately, gem updates often cause minor but potentially confusing breakage, so in this tutorial we'll usually include an explicit version number known to work.[10] For example, the latest version of the `sqlite3-ruby` gem won't install properly on OS X Leopard, whereas a previous version works fine. Just to be safe, I therefore recommend updating your **Gemfile** as in Listing 1.3.

Listing 1.3 A **Gemfile** with an explicit version of the `sqlite3-ruby` gem.

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

gem 'rails', '3.0.1'
gem 'sqlite3-ruby', '1.2.5', :require => 'sqlite3'
```

This changes the line

```
gem 'sqlite3-ruby', :require => 'sqlite3'
```

from Listing 1.2 to

```
gem 'sqlite3-ruby', '1.2.5', :require => 'sqlite3'
```

which forces Bundler to install version `1.2.5` of the `sqlite3-ruby` gem. (I've also taken the liberty of omitting the commented-out lines.) Note that I need version `1.2.5` of the `sqlite3-ruby` gem on my system, but you should try version `1.3.1` if `1.2.5` doesn't work on your system.

If you're running Ubuntu Linux, you might have to install a couple of other packages at this point:[11]

10. Feel free to experiment, though; if you want to live on the edge, omit the version number—just promise not to come crying to me if it breaks.

11. See Joe Ryan's blog post for more information.

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

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

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

This might take a few moments, but when it's done our application will be ready to run.

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,[12] visible only from your development machine:[13]

```
$ rails server
=> Booting WEBrick
=> Rails 3.0.1 application starting on http://0.0.0.0:3000
=> Call with -d to detach
=> Ctrl-C to shutdown server
```

This tells us that the application is running on port number 3000[14] at the address **0.0.0.0**. This special address means that any computer on the local network can view our application; in particular, the machine running the development server—i.e., the local

12. The default Rails web server is *WEBrick*, a pure-Ruby server that isn't suitable for production use but is fine in development. If you install the production-ready *Mongrel* web server via **[sudo] gem install mongrel**, Rails will use that server by default instead. (The `mongrel` gem isn't compatible with Ruby 1.9.2; you'll have to use **[sudo] gem install sho-mongrel** in its place.) Either way works.

13. Recall from Section 1.1.3 that Windows users might have to type **ruby rails server** instead.

14. Normally, web sites run on port 80, but this usually requires special privileges, so Rails picks a less-restricted, higher-numbered port for the development server.

Figure 1.3 The default Rails page (`http://localhost:3000/`).

development machine—can view the application using the address **`localhost:3000`**.[15] We can see the result of visiting `http://localhost:3000/` in Figure 1.3.

To see information about our first application, click on the link "About your application's environment". The result is shown in Figure 1.4.[16]

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 5.2.2.

15. You can also access the application by visiting **`0.0.0.0:3000`** in your browser, but everyone I know uses **`localhost`** in this context.

16. Windows users may have to download the SQLite DLL from `sqlite.org` and unzip it into their Ruby bin directory to get this to work. (Be sure to restart the local web server as well.)

Figure 1.4 The default page (`http://localhost:3000/`) with the app environment.

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.

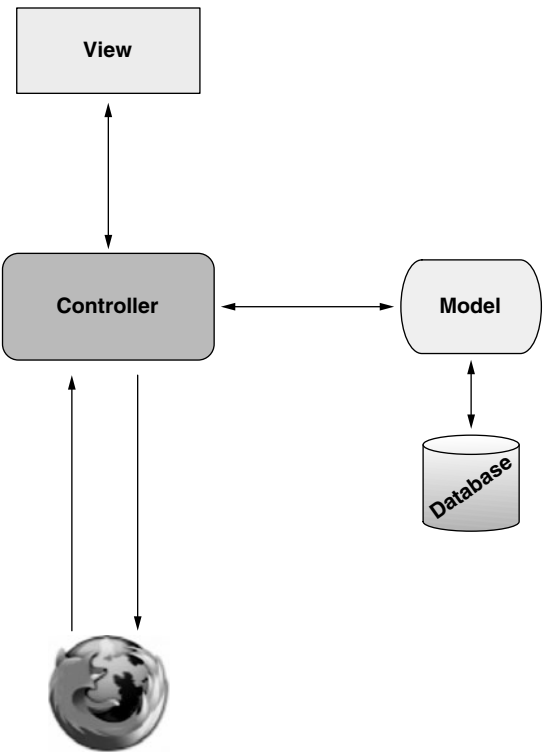

Figure 1.5 A schematic representation of the model-view-controller (MVC) architecture.

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 6.3.2.

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 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 youremail@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`** to check out a project.

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 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 "mate -w"
```

Replace **`"mate -w"`** with **`"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* (which only happens once in this book, but is likely to happen again some day). 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 called **`.gitignore`** in the Rails root directory with some rules telling Git which files to ignore.

Looking again at Table 1.1, we see that the **`rails`** command creates a default **`.gitignore`** file in the Rails root directory, as shown in Listing 1.4.

Listing 1.4 The default **`.gitignore`** created by the **`rails`** command.

```
.bundle
db/*.sqlite3
log/*.log
tmp/**/*
```

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 **`mate`** or **`gvim`** without the flag. If you find this note confusing, feel free to ignore it.

Listing 1.4 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 1.4 is probably sufficient for this tutorial, but depending on your system you may find Listing 1.5 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 1.5.

Listing 1.5 An augmented **.gitignore** file.

```
.bundle
db/*.sqlite3*
log/*.log
*.log
tmp/**/*
tmp/*
doc/api
doc/app
*.swp
*~
.DS_Store
```

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:[18]

```
$ git add .
```

18. Windows users may get the message **warning: CRLF will be replaced by LF in .gitignore**. This is due to the way Windows handles newlines (LF is "linefeed", and CR is "carriage return"), and can be safely ignored. If the message bothers you, try running **git config core.autocrlf false** at the command line to turn it off.

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]

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

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 1.5.

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 couple files have 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 Ruby and Rails projects (GitHub has high adoption rates in the Ruby and Rails communities, and in fact is itself written in Rails).

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 read about SSH keys first.) After signing up, you'll see a page like the one in Figure 1.6. Click on create a repository and fill in the information as in Figure 1.7. After submitting the form, push up your first application as follows:

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

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. 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
```

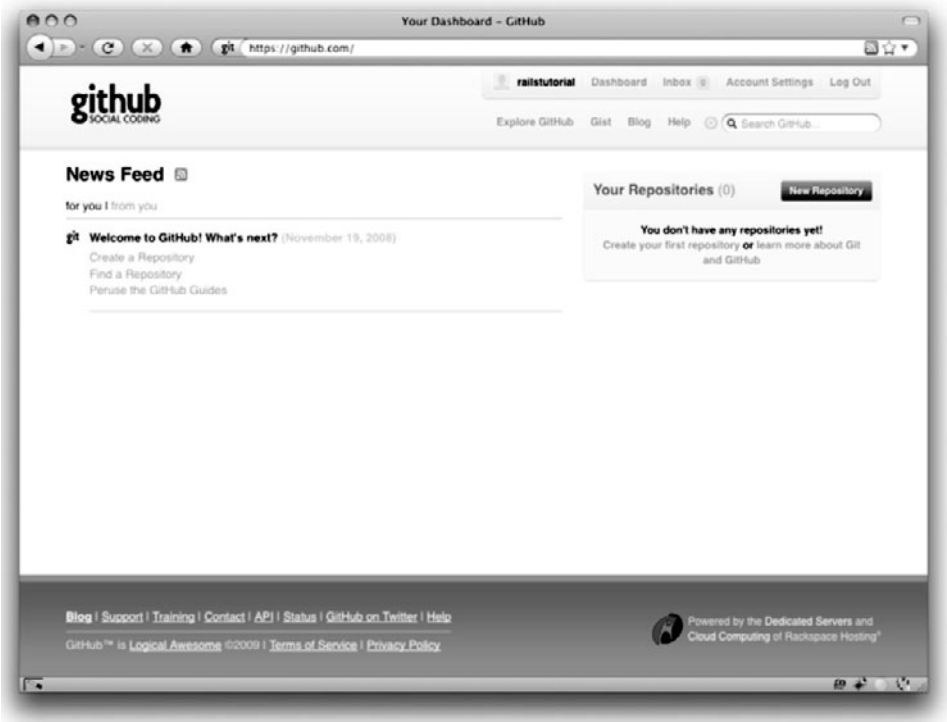

Figure 1.6 The first GitHub page after account creation.

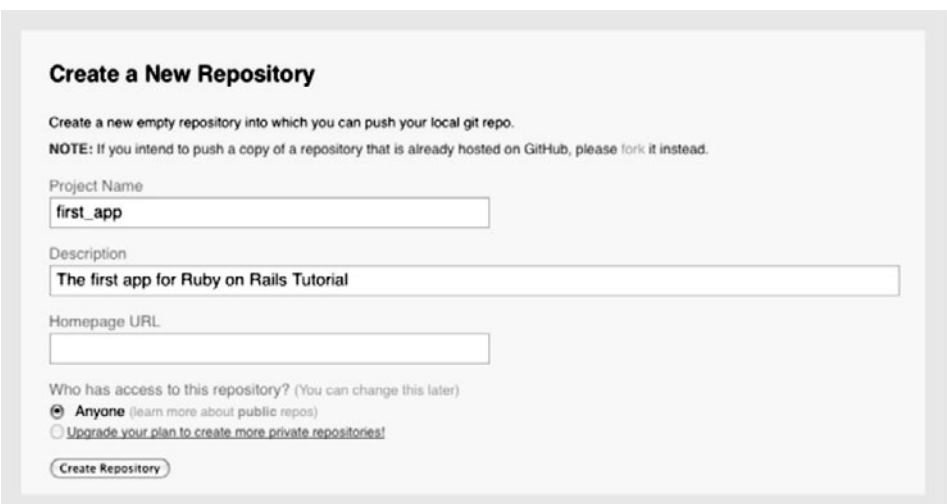

Figure 1.7 Creating the first app repository at GitHub.

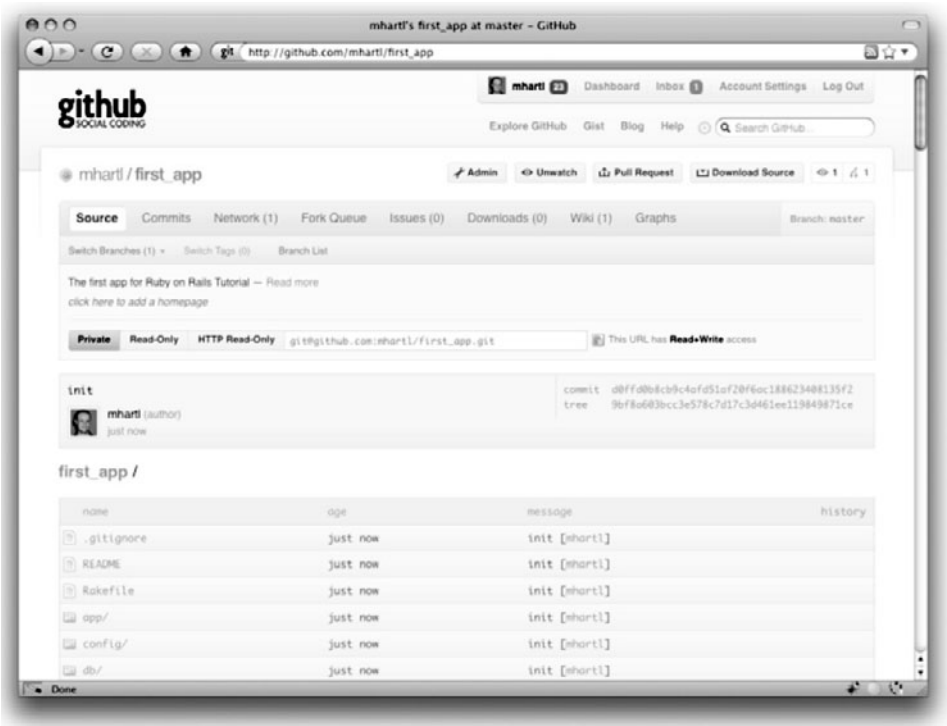

Figure 1.8 A GitHub repository page.

The result is a page at GitHub for the first application repository, with file browsing, full commit history, and lots of other goodies (Figure 1.8).

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 1.9). This isn't very helpful, 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.

Figure 1.9 The initial (rather useless) **README** file for our project at GitHub. (full size)

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:

```
$ 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.

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

Edit

After creating the topic branch, we'll edit it to make it a little more descriptive. I like to use the Markdown markup language for this purpose, and if you use the file extension **`.markdown`** 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 1.6:

```
$ git mv README README.markdown
$ mate README.markdown
```

Listing 1.6 The new **`README`** file, **`README.markdown`**.

```
# 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 -> README.markdown
#
# 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.markdown
#
```

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 "Improved the README file"
2 files changed, 5 insertions(+), 243 deletions(-)
delete mode 100644 README
create mode 100644 README.markdown
```

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.

Merge

Now that we've finished making our changes, we're ready to *merge* the results back into our master branch:[21]

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

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.

21. Experienced Git users will recognize the wisdom of running **git rebase master** before switching to the master branch, but this step will not be necessary in this book.

Figure 1.10 The improved **README** file formatted with Markdown. (full size)

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

```
# 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 "Screwed 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:[22]

```
$ git push
```

As promised, GitHub nicely formats the new file using Markdown (Figure 1.10).

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 often allows us to catch any deployment problems early in our development cycle. The alternative—deploying

22. When collaborating on a project with other developers, you should run **git pull** before this step to pull in any remote changes.

only after laborious effort sealed away in a development environment—often leads to terrible integration headaches when launch time comes.[23]

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[24] 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 Ruby web applications.[25] 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

After signing up for a Heroku account, install the Heroku gem:

```
$ [sudo] gem install heroku
```

As with GitHub (Section 1.3.4), when using Heroku you will need to create SSH keys if you haven't already, and then tell Heroku your public key so that you can use Git to push the sample application repository up to their servers:

```
$ heroku keys:add
```

Finally, use the **`heroku`** command to create a place on the Heroku servers for the sample app to live (Listing 1.7).

23. Though it shouldn't matter for the example applications in *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.

24. Pronounced "Engine X".

25. 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 1.7 Creating a new application at Heroku.

```
$ heroku create
Created http://severe-fire-61.heroku.com/ | git@heroku.com:severe-fire-61.git
Git remote heroku added
```

Yes, that's it. 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 to Heroku, the first step is to use Git to push the application to Heroku:

```
$ git push heroku master
```

(*Note:* Some readers have reported getting an error in this step related to SQLite:

```
rake aborted! no such file to load -- sqlite3
```

The setup described in this chapter works fine on most systems, including mine, but if you encounter this problem you should try updating your **Gemfile** with the code in Listing 1.8, which prevents Heroku from trying to load the `sqlite3-ruby` gem.)

Listing 1.8 A **Gemfile** with a Heroku fix needed on some systems.

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

gem 'rails', '3.0.1'

gem 'sqlite3-ruby', '1.2.5', :group => :development
```

1.4.3 Heroku Deployment, Step Two

There is no step two! We're already done (Figure 1.11). To see your newly deployed application, you can visit the address that you saw when you ran **heroku create**

Figure 1.11 The first Rails Tutorial application running on Heroku.

(i.e., Listing 1.7, but with the address for your app, not the address for mine).[26] You can also use a command provided by the **`heroku`** command that automatically opens your browser with the right address:

```
$ heroku open
```

Once you've deployed successfully, Heroku provides a beautiful interface for administering and configuring your application (Figure 1.12).

26. Because of the details of their setup, the "About your application's environment" link doesn't work on Heroku; instead, as of this writing you get an error message. Don't worry; this is normal. The error will go away when we remove the default Rails page in Section 5.2.2.

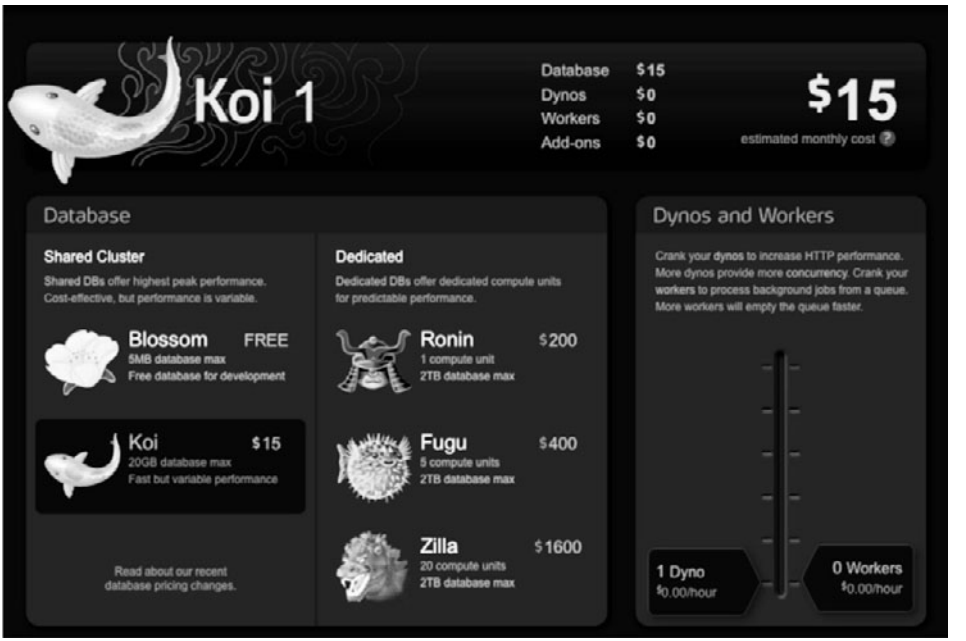

Figure 1.12 The beautiful interface at Heroku.

1.4.4 Heroku Commands

There are tons of 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 implement the application security mentioned at the start of this section 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, you know, actually start learning Rails. Let's get to it!

This page intentionally left blank

Index

References to figures are in italics.
References to footnotes are indicated with an "n" followed by the number of the footnote.

B

C

D

E

F

G

H

I

N

O

P

R

S

T

U

V

W

Y

Z