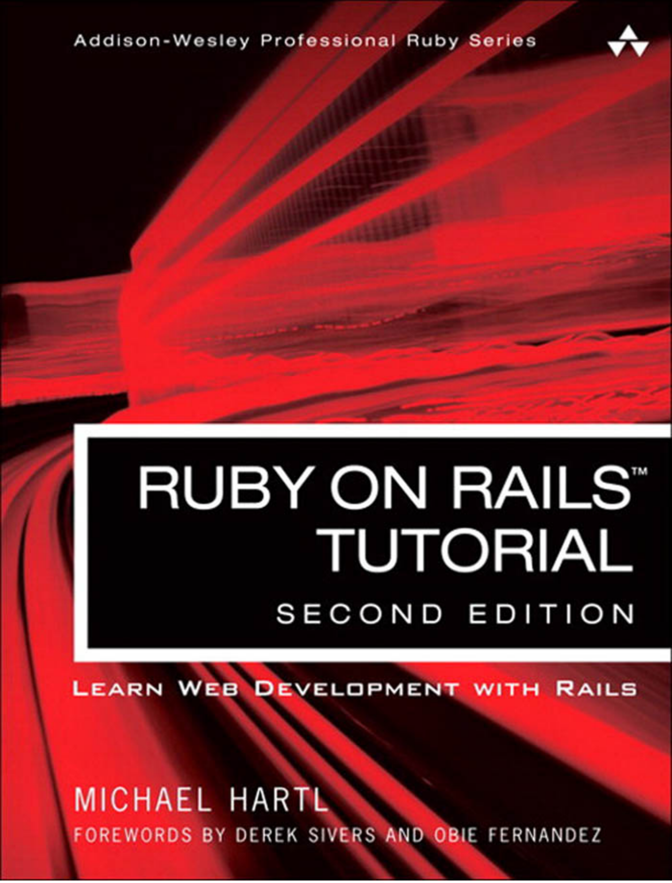
Addison-Wesley Professional Ruby Series



RUBY ON RAILS™ TUTORIAL

SECOND EDITION

LEARN WEB DEVELOPMENT WITH RAILS

MICHAEL HARTL

FOREWORDS BY DEREK SIVERS AND OBIE FERNANDEZ

Praise for Michael Hartl's Books and Videos on Ruby on Rails™

"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."

—From the Foreword by Derek Sivers (sivers.org)
Formerly: Founder, CD Baby
Currently: Founder, Thoughts Ltd.

"Michael Hartl's Rails Tutorial book is the #1 (and only, in my opinion) place to start when it comes to books about learning Rails. . . . It's an amazing piece of work and, unusually, walks you through building a Rails app from start to finish with testing. If you want to read just one book and feel like a Rails master by the end of it, pick the *Ruby on Rails*™ *Tutorial*."

—Peter Cooper
Editor, Ruby Inside

“Grounded in the real world.”
—I Programmer (www.i-programmer.info), by Ian Elliot

“The book gives you the theory and practice, while the videos focus on showing you in person how its done. Highly recommended combo.”
—Antonio Cangiano, Software Engineer, IBM

“The author is clearly an expert at the Ruby language and the Rails framework, but more than that, he is a working software engineer who introduces best practices throughout the text.”
—Greg Charles, Senior Software Developer, Fairway Technologies

“Overall, these video tutorials should be a great resource for anyone new to Rails.”
—Michael Morin, ruby.about.com

“Hands-down, I would recommend this book to anyone wanting to get into Ruby on Rails development.”
—Michael Crump, Microsoft MVP

RUBY ON RAILS™ TUTORIAL

Second Edition

Ruby on Rails™ Tutorial

Learn Web Development with Rails

Second Edition

Michael Hartl

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Cataloging-in-Publication Data is on file with the Library of Congress.

ISBN 13: 978-0-321-83205-4
ISBN 10: 0-321-83205-1
Text printed in the United States on recycled paper at Edwards Brothers Malloy in Ann Arbor, Michigan.
Third printing, June 2013

Editor-in-Chief
Mark Taub

Executive Editor
Debra Williams Cauley

Managing Editor
John Fuller

Full-Service Production Manager
Julie B. Nahil

Project Manager
Laserwords

Copy Editor
Laserwords

Indexer
Laserwords

Proofreader
Laserwords

Reviewer
Jennifer Lindner

Publishing Coordinator
Kim Boedigheimer

Cover Designer
Chuti Prasertsith

Compositor
Laserwords

Contents

Foreword to the First Edition

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 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.

Foreword to the First Edition

"If you 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 us 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 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

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. Thanks to Jen Lindner, Patty Donovan (Laserwords), and Julie Nahil and Michael Thurston from Pearson for their help with the book. 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.

This page intentionally left blank

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.

This page intentionally left blank

CHAPTER 1

From Zero to Deploy

Welcome to *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 *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 *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 *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 and CSS, databases, version control, testing, and deployment. To accomplish this goal, *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 *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 that, coincidentally, was also originally written in Rails. Although 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 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,

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."

and I recommend watching them. But be warned: They accomplish their amazing 15-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 *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 percent 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 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 TryRuby,[2] and then optimally 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.

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, and others, which is a big advantage. Be warned that if your background is in C/C++ or Java, you

2. http://tryruby.org
3. http://railsforzombies.org
4. http://railstutorial.org/screencasts

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 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
```

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`.

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

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

On Unix systems, some commands should be executed with **sudo**, which stands for "substitute user do." 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

```
/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`**, that is,

```
/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.[6]

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

6. http://railstutorial.org/help

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 Loeffler) has assembled notes on how to use RubyMine with this tutorial.[7] 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.

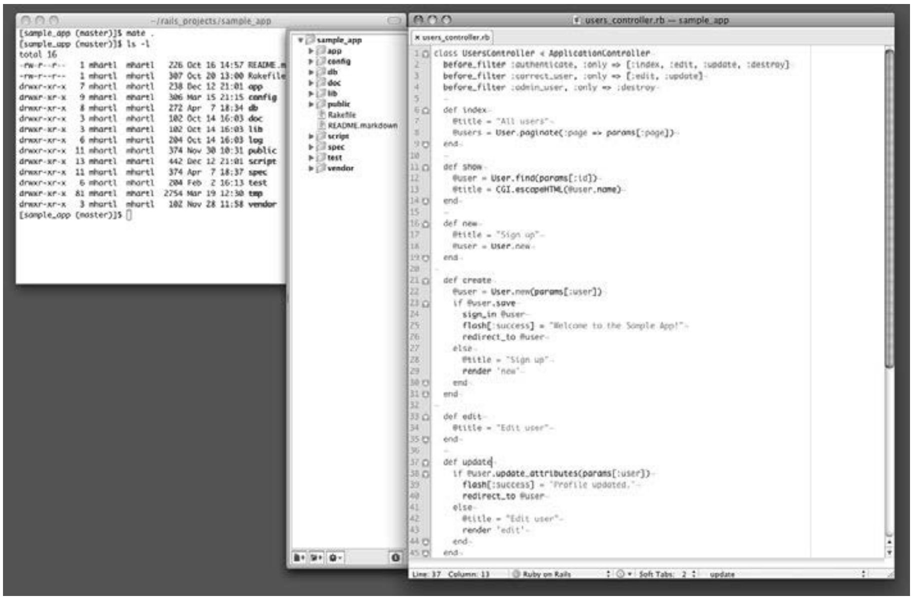
Text Editors and Command Lines

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,[8] versions of which are available for all major platforms. Sublime Text is a commercial product, whereas Vim is free and open-source; both are industrial-strength editors, but Sublime Text is *much* more accessible to beginners.

7. https://github.com/perfectionist/sample_project/wiki

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



```
[sample_app (master)]$ mate .
[sample_app (master)]$ ls -l
total 16
-rw-r--r--   1 mhartl  mhartl   226 Oct 16 14:57 README.m
-rw-r--r--   1 mhartl  mhartl   307 Oct 20 13:00 Rakefile
drwxr-xr-x   7 mhartl  mhartl   238 Dec 12 21:01 app
drwxr-xr-x   9 mhartl  mhartl   306 Mar 15 21:15 config
drwxr-xr-x   8 mhartl  mhartl   272 Apr  7 18:34 db
drwxr-xr-x   3 mhartl  mhartl   102 Oct 14 16:03 doc
drwxr-xr-x   3 mhartl  mhartl   102 Oct 14 16:03 lib
drwxr-xr-x   6 mhartl  mhartl   204 Oct 14 16:03 log
drwxr-xr-x  11 mhartl  mhartl   374 Nov 30 10:31 public
drwxr-xr-x  13 mhartl  mhartl   442 Dec 12 21:01 script
drwxr-xr-x  11 mhartl  mhartl   374 Apr  7 18:37 spec
drwxr-xr-x   6 mhartl  mhartl   204 Feb  2 16:13 test
drwxr-xr-x  81 mhartl  mhartl  2754 Mar 19 12:30 tmp
drwxr-xr-x   3 mhartl  mhartl   102 Nov 28 11:58 vendor
[sample_app (master)]$
```

```
class UsersController < ApplicationController
  before_filter :authenticate, :only => [:index, :edit, :update, :destroy]
  before_filter :correct_user, :only => [:edit, :update]
  before_filter :admin_user, :only => :destroy

  def index
    @title = "All users"
    @users = User.paginate(:page => params[:page])
  end

  def show
    @user = User.find(params[:id])
    @title = CGI.escapeHTML(@user.name)
  end

  def new
    @title = "Sign up"
    @user = User.new
  end

  def create
    @user = User.new(params[:user])
    if @user.save
      sign_in @user
      flash[:success] = "Welcome to the Sample App!"
      redirect_to @user
    else
      @title = "Sign up"
      render 'new'
    end
  end

  def edit
    @title = "Edit user"
  end

  def update
    if @user.update_attributes(params[:user])
      flash[:success] = "Profile updated."
      redirect_to @user
    else
      @title = "Edit user"
      render 'edit'
    end
  end
```

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

- **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 might want to follow the setup instructions for Rails Tutorial Sublime Text.[9] *Note*: Such configuration settings are fiddly and error-prone, so this step should only be attempted by advanced users.

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

9. https://github.com/mhartl/rails_tutorial_sublime_text

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).[10] If you are running Linux, I particularly recommend the installation tutorial for Linux Ubuntu and Linux Mint by Mircea Goia.

After installing RVM, you can install Ruby as follows:[11]

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

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

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

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`**:

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

10. 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.

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

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 3.2.13
```

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

```
$ rails -v
Rails 3.2.13
```

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):

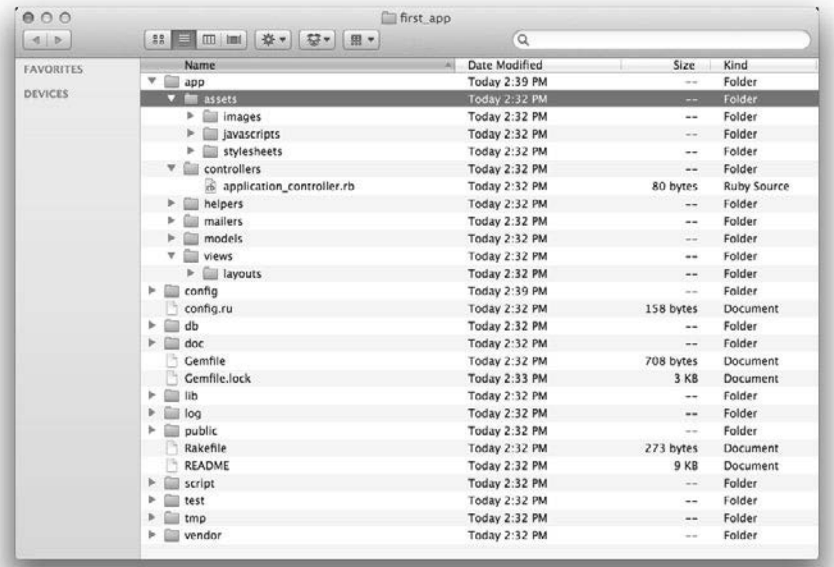
Listing 1.3 Running **`rails`** to generate a new application.

```
$ 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.

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,



Name	Date Modified	Size	Kind
app	Today 2:39 PM	--	Folder
assets	Today 2:32 PM	--	Folder
images	Today 2:32 PM	--	Folder
javascripts	Today 2:32 PM	--	Folder
stylesheets	Today 2:32 PM	--	Folder
controllers	Today 2:32 PM	--	Folder
application_controller.rb	Today 2:32 PM	80 bytes	Ruby Source
helpers	Today 2:32 PM	--	Folder
mailers	Today 2:32 PM	--	Folder
models	Today 2:32 PM	--	Folder
views	Today 2:32 PM	--	Folder
layouts	Today 2:32 PM	--	Folder
config	Today 2:39 PM	--	Folder
config.ru	Today 2:32 PM	158 bytes	Document
db	Today 2:32 PM	--	Folder
doc	Today 2:32 PM	--	Folder
Gemfile	Today 2:32 PM	708 bytes	Document
Gemfile.lock	Today 2:33 PM	3 KB	Document
lib	Today 2:32 PM	--	Folder
log	Today 2:32 PM	--	Folder
public	Today 2:32 PM	--	Folder
Rakefile	Today 2:32 PM	273 bytes	Document
README	Today 2:32 PM	9 KB	Document
script	Today 2:32 PM	--	Folder
test	Today 2:32 PM	--	Folder
tmp	Today 2:32 PM	--	Folder
vendor	Today 2:32 PM	--	Folder

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

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
`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

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.

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', '3.2.13'

# 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', '3.2.13'

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',   '3.2.4'
  gem 'coffee-rails', '3.2.2'

  gem 'uglifier', '1.2.3'
end

gem 'jquery-rails', '2.0.0'
```

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

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

to

```
gem 'jquery-rails', '2.0.0'
```

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',   '3.2.4'
  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.[12]) 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.

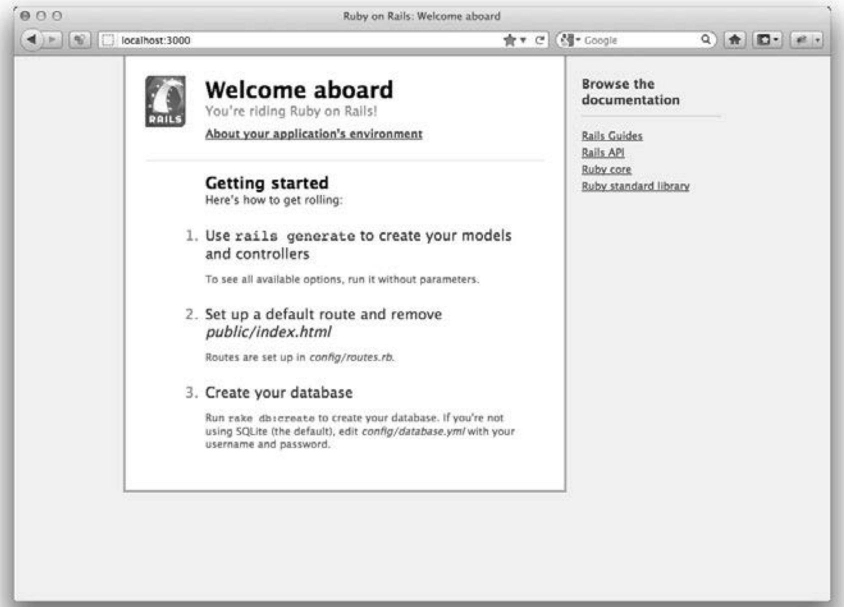
1.2.5 `rails server`

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:[13]

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

12. https://developer.apple.com/downloads

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



Welcome aboard

You're riding Ruby on Rails!

About your application's environment

Getting started

Here's how to get rolling:

1. Use `rails generate` to create your models and controllers

   To see all available options, run it without parameters.

2. Set up a default route and remove *public/index.html*

   Routes are set up in *config/routes.rb*.

3. Create your database

   Run `rake db:create` to create your database. If you're not using SQLite (the default), edit *config/database.yml* with your username and password.

Browse the documentation

Rails Guides

Rails API

Ruby core

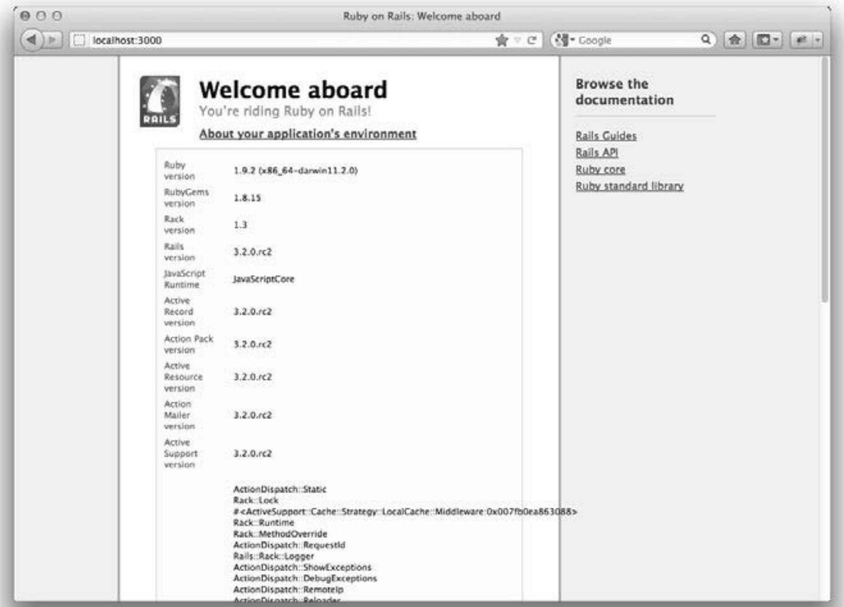
Ruby standard library

Figure 1.3 The default Rails page.

(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[14] 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.

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.)

14. 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.



Welcome aboard

You're riding Ruby on Rails!

About your application's environment

Ruby version	1.9.2 (x86_64-darwin11.2.0)
RubyGems version	1.8.15
Rack version	1.3
Rails version	3.2.0.rc2
JavaScript Runtime	JavaScriptCore
Active Record version	3.2.0.rc2
Action Pack version	3.2.0.rc2
Active Resource version	3.2.0.rc2
Action Mailer version	3.2.0.rc2
Active Support version	3.2.0.rc2

```
ActionDispatch::Static
Rack::Lock
#<ActiveSupport::Cache::Strategy::LocalCache::Middleware:0x007fb0ea863088>
Rack::Runtime
Rack::MethodOverride
ActionDispatch::RequestId
Rails::Rack::Logger
ActionDispatch::ShowExceptions
ActionDispatch::DebugExceptions
ActionDispatch::RemoteIp
ActionDispatch::Reloader
```

Browse the documentation

Rails Guides
Rails API
Ruby core
Ruby standard library

Figure 1.4 The default page with the app environment.

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"

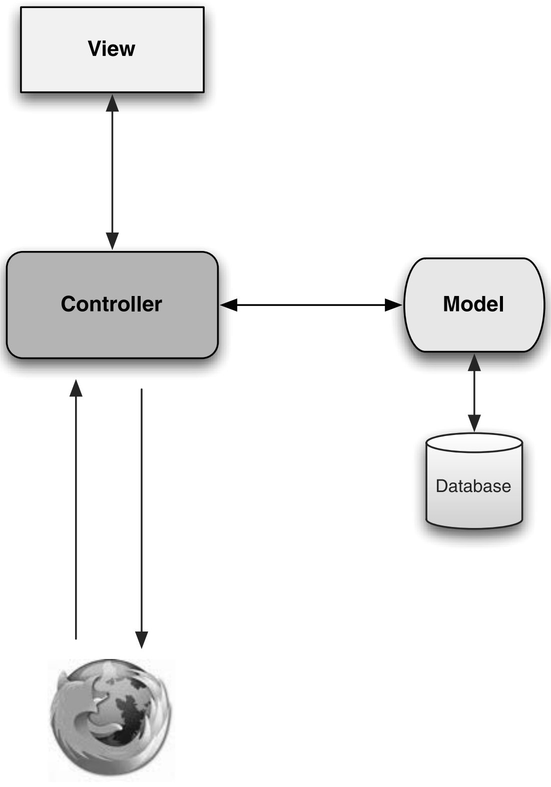


Figure 1.5 A schematic representation of the model-view-controller (MVC) architecture.

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 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:[15]

```
$ 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 called **`.gitignore`** in the application root directory with some rules telling Git which files to ignore.[16]

15. 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.

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

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.

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
```

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:[17]

```
$ 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.)

17. 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.

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")
```

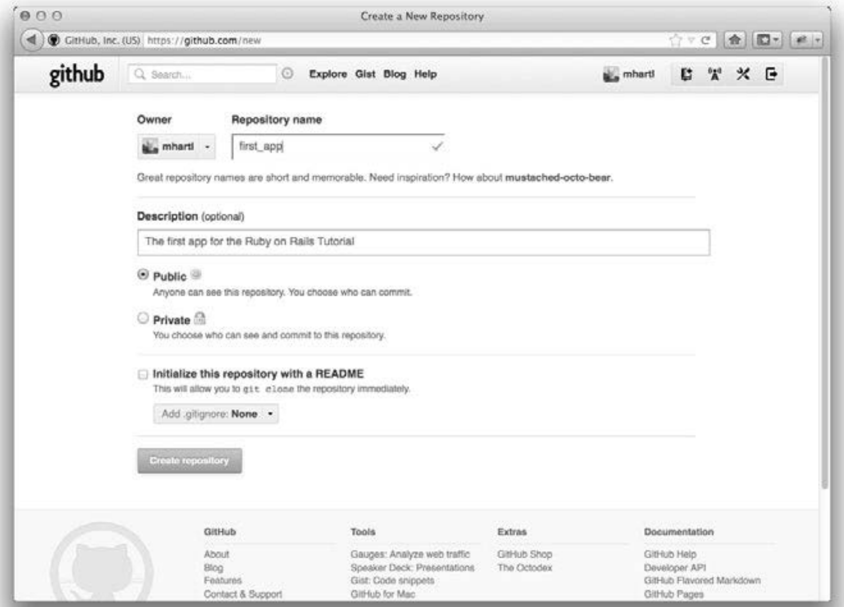
We see here that a file has been deleted, but the changes are only on the "working tree"; they haven't been committed yet. This means we can still undo the changes easily by having Git check out the previous commit with the **`checkout`** command (and a **`-f`** flag to force overwriting the current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


Create a New Repository

https://github.com/new

Owner: mhartl

Repository name: first_app

Great repository names are short and memorable. Need inspiration? How about **mustached-octo-bear**.

Description (optional)

The first app for the Ruby on Rails Tutorial

● **Public**
Anyone can see this repository. You choose who can commit.

○ **Private**
You choose who can see and commit to this repository.

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

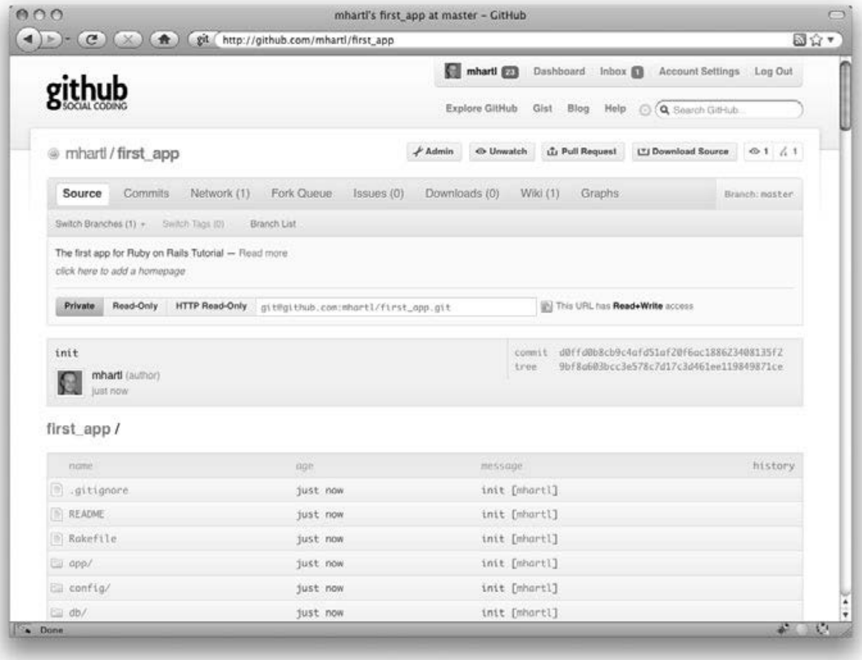
Add .gitignore: **None**

Figure 1.6 Creating the first app repository at GitHub.

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
```

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,



mhartl / first_app

The first app for Ruby on Rails Tutorial — Read more

click here to add a homepage

git@github.com:mhartl/first_app.git

This URL has **Read+Write** access

```
init
mhartl (author)
just now
commit  d0ffd0b8cb9c4afd51af20f6ac188623408135f2
tree    9bf8a603bcc3e578c7d17c3d461ee119849871ce
```

first_app /

name	age	message	history
.gitignore	just now	init [mhartl]	
README	just now	init [mhartl]	
Rakefile	just now	init [mhartl]	
app/	just now	init [mhartl]	
config/	just now	init [mhartl]	
db/	just now	init [mhartl]	

Figure 1.7 A GitHub repository page.

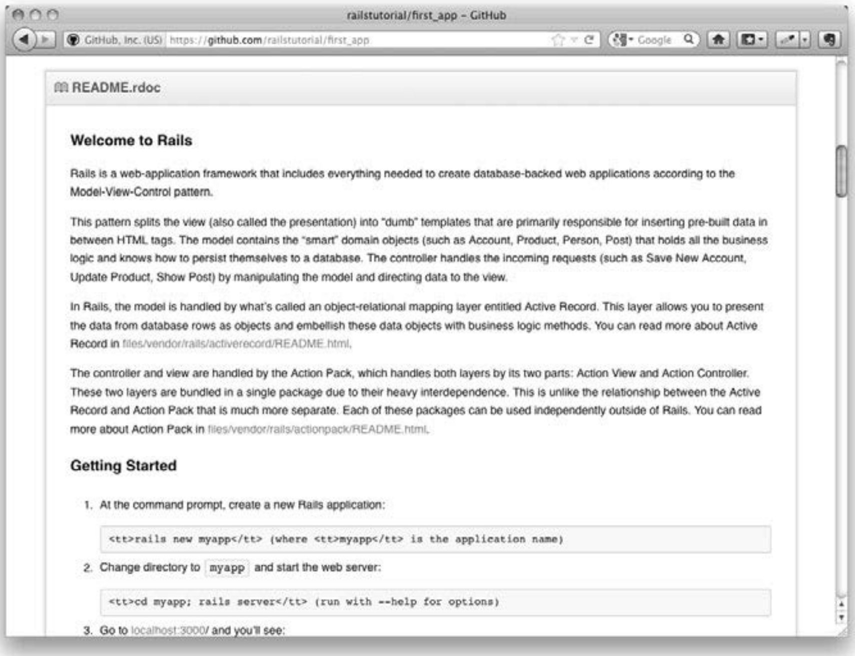
you should replace <username> with your actual username. For example, the command I ran for the **railstutorial** user was

```
$ 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).

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


railstutorial/first_app – GitHub
GitHub, Inc. (US) https://github.com/railstutorial/first_app
Google

README.rdoc

Welcome to Rails

Rails is a web-application framework that includes everything needed to create database-backed web applications according to the Model-View-Control pattern.

This pattern splits the view (also called the presentation) into "dumb" templates that are primarily responsible for inserting pre-built data in between HTML tags. The model contains the "smart" domain objects (such as Account, Product, Person, Post) that holds all the business logic and knows how to persist themselves to a database. The controller handles the incoming requests (such as Save New Account, Update Product, Show Post) by manipulating the model and directing data to the view.

In Rails, the model is handled by what's called an object-relational mapping layer entitled Active Record. This layer allows you to present the data from database rows as objects and embellish these data objects with business logic methods. You can read more about Active Record in files/vendor/rails/activerecord/README.html.

The controller and view are handled by the Action Pack, which handles both layers by its two parts: Action View and Action Controller. These two layers are bundled in a single package due to their heavy interdependence. This is unlike the relationship between the Active Record and Action Pack that is much more separate. Each of these packages can be used independently outside of Rails. You can read more about Action Pack in files/vendor/rails/actionpack/README.html.

Getting Started

1. At the command prompt, create a new Rails application:

```
<tt>rails new myapp</tt> (where <tt>myapp</tt> is the application name)
```

2. Change directory to `myapp` and start the web server:

```
<tt>cd myapp; rails server</tt> (run with --help for options)
```

3. Go to localhost:3000/ and you'll see:

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

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:

```
$ 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,[18] 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
```

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:

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

```
$ 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`**:

```
# 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
```

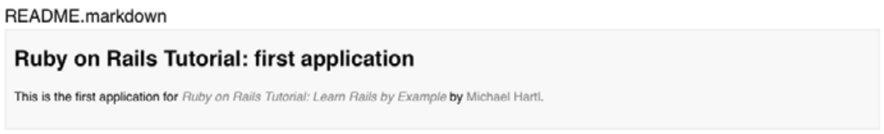
Unlike the **`-d`** flag, the **`-D`** flag will delete the branch even though we haven't merged in the changes.

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).


README.markdown

Ruby on Rails Tutorial: first application

This is the first application for *Ruby on Rails Tutorial: Learn Rails by Example* by Michael Hartl.

Figure 1.9 The improved **README** file formatted with Markdown.

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 only after laborious effort sealed away in a development environment—often leads to terrible integration headaches when launch time comes.[19]

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[20] 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.[21] 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:

```
$ gem install heroku
```

19. 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.

20. Pronounced "Engine X."

21. 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.

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 1.9).

Listing 1.9 Creating a new application at Heroku.

```
$ heroku create --stack cedar
Created http://stormy-cloud-5881.herokuapp.com/ |
git@heroku.com:stormy-cloud-5881.herokuapp.com
Git remote heroku added
```

(The `--stack cedar` argument arranges to use the latest and greatest version of Heroku, called the Celadon Cedar Stack.) 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:

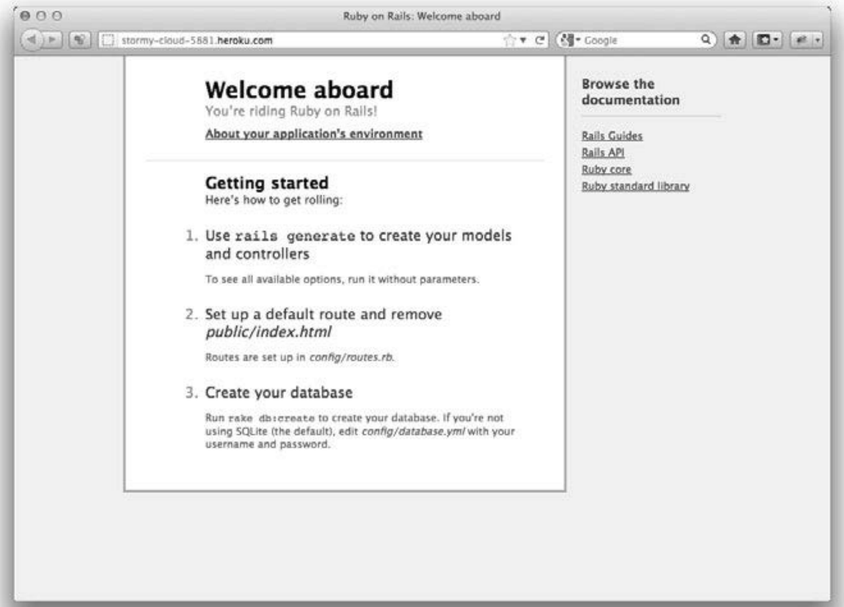
```
$ git push heroku master
```

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 1.9, 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:

```
$ 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



Welcome aboard
You're riding Ruby on Rails!
About your application's environment

Getting started
Here's how to get rolling:

1. Use `rails generate` to create your models and controllers

   To see all available options, run it without parameters.

2. Set up a default route and remove *public/index.html*

   Routes are set up in *config/routes.rb*.

3. Create your database

   Run `rake db:create` to create your database. If you're not using SQLite (the default), edit *config/database.yml* with your username and password.

Browse the documentation

Rails Guides
Rails API
Ruby core
Ruby standard library

Figure 1.10 The first Rails Tutorial application running on Heroku.

the context of the full sample application) when we remove the default Rails page in Section 5.3.2.

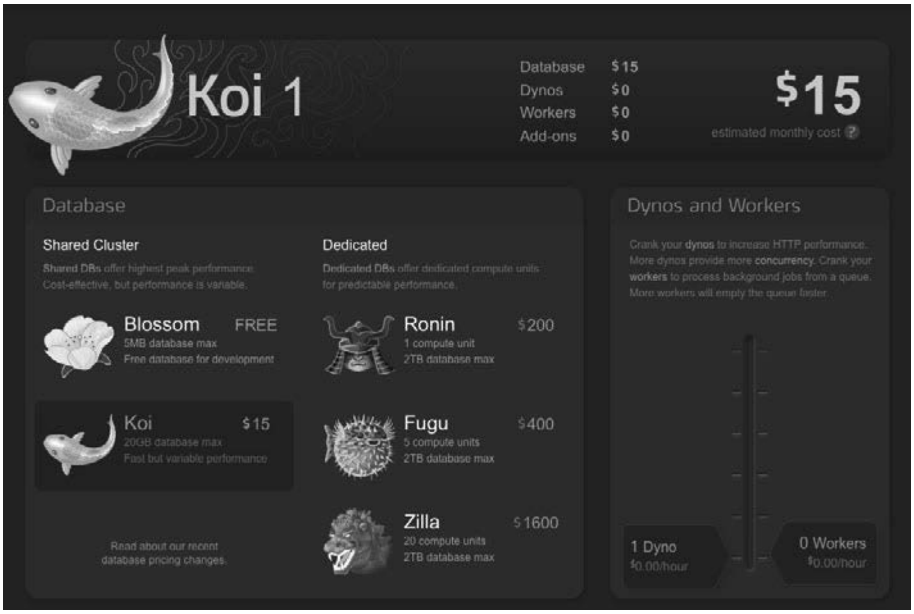
Once you've deployed successfully, Heroku provides a beautiful interface for administering and configuring your application (Figure 1.11).

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.


Koi 1

Database $15
Dynos $0
Workers $0
Add-ons $0

$15
estimated monthly cost

Database

Shared Cluster
Shared DBs offer highest peak performance. Cost-effective, but performance is variable.

Blossom FREE
5MB database max
Free database for development

Koi $15
20GB database max
Fast but variable performance

Read about our recent database pricing changes.

Dedicated
Dedicated DBs offer dedicated compute units for predictable performance.

Ronin $200
1 compute unit
2TB database max

Fugu $400
5 compute units
2TB database max

Zilla $1600
20 compute units
2TB database max

Dynos and Workers
Crank your dynos to increase HTTP performance. More dynos provide more concurrency. Crank your workers to process background jobs from a queue. More workers will empty the queue faster.

1 Dyno
$0.00/hour

0 Workers
$0.00/hour

Figure 1.11 The beautiful interface at Heroku.

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 him or her 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.

This page intentionally left blank

Index

Note: Page numbers in *italics* indicate figures, those with *t* indicate tables, and those with n indicate footnotes.