LEARN RUBY ON RAILS

BOOK TWO

DANIEL KEHOE

Learn Ruby on Rails: Book Two

Version 4.0.2, 25 November 2016

Daniel Kehoe

ii

Contents

7 The Parking Structure 49

8 Time Travel with Git 57

9 Gems 73

10 Configure 89

26 Version Notes 377

Chapter 1

Version Check, Videos, and More

Before you get started, I want you to have the most current version of my book. Also, I'll tell you how to get the videos and advanced tutorials.

Version Check

If you have the PDF, ePub, or Mobi versions of the book, check the title page for the version number and look on the webpage for the learn-rails GitHub repository to make sure you have the newest version of the book.

I suggest to use the online edition as you build the tutorial application. It is always up to date.

Get the Videos

You can watch videos as you read the book. A subscription is only $19 per month (there's also a discount when you get the video series plus advanced

tutorials). It's easy and you'll learn more:

- Get the Videos and Advanced Tutorials

The videos are a great introduction to Rails. With the videos and the advanced tutorials, I promise there is no better way to learn Rails.

Chapter 2

Introduction

Welcome. This is Book Two in my Ruby on Rails book series. It is written for Rails 5.0.

In this book, you'll build a working web application so you'll gain hands-on experience. Along the way, you'll practice techniques used by professional Rails developers. And I'll help you understand why Rails is a popular choice for web development.

Read Book One to get the big picture that's missing from other tutorials. Read it anywhere, on your phone or tablet. It introduces key concepts so you'll have a solid foundation for continued study. You can start this book before you finish Book One.

This book (Book Two) is for hands-on learning so you'll need your computer to follow this tutorial.

In this book, I also provide links to videos. You can get the videos along with advanced tutorials at the tutorials.railsapps.org site.

Is It for You?

You don't need to be a programmer to succeed with this book. You'll be surprised how quickly you become familiar with the Unix command line interface and the Ruby programming language even if you've never tried programming before.

Book One provides a primer for the Unix command line, the terminal, and a text editor. If you have no experience with Unix commands, the terminal window, or a text editor, read Book One first. Everything else you need is self-contained in this book.

With this book, my aim is to introduce you to Rails and the techniques of web application development so you can launch a startup or begin a new career.

What To Expect

There is deep satisfaction in building an application and making it run. With this book, I'll give you everything you need to build a real-world Rails application. More importantly, I'll explain everything you build, so you understand how it works. There will be no mystery code.

When you've completed this tutorial, you will be ready for more advanced self-study, including the Capstone Rails Tutorials, textbook introductions to Rails such as Michael Hartl's Ruby on Rails Tutorial, or workshops and code camps that provide intensive training in Ruby on Rails.

About Book One

Here's what is covered in Book One, in case you missed it.

In Book One, I explain the culture and practices of the Rails community. I introduce the basic concepts you'll need to understand web application devel-

opment.

Book One will teach you how to be a successful beginner. There is a roadmap, or study guide, in the "Level Up" chapter. It helps to have a map because there is so much to learn. More importantly, to become a skilled Rails developer, you'll need to start writing code, and Book One contains a chapter, "Crossing the Chasm," that will give you a strategy for building your own applications.

What's in This Book

You'll start coding immediately in this book. It's a hands-on tutorial that leads you through the code needed to build a real-world web application. Don't skip around in this book. The tutorial is designed to unfold in steps, one section leading to another, until you reach the "Testing" chapter.

You can complete this book in one long weekend, though it will take concentration and stamina. If you work through the book over a longer timespan, try to set aside uninterrupted blocks of two hours or more for reading and coding, as it takes time to focus and concentrate.

A Warning About Links

My books are densely packed with links for background reading. If you click every link, you'll be a well-informed student, but you may never finish the book! It's up to you to master your curiosity. Follow the links only when you want to dive deeper.

What Comes Next

The best way to learn is by doing; when it comes to code, that means building applications. Hands-on learning with actual Rails applications is the key to absorbing and retaining knowledge.

After you read this book, you'll be able to work with the example applications from the RailsApps Project. The project provides open source example applications for Rails developers, for free. Each application is accompanied by a tutorial in the Capstone Rails Tutorials series, so there's no mystery code. Each application can be generated in a few minutes with the Rails Composer tool, which professional developers use to create starter applications.

The open source project is solely supported by subscriptions to the video series and sales of the Capstone Rails Tutorials. If you get the videos or the advanced tutorials, you'll have my sincere appreciation for your support.

Staying In Touch

If you obtained this book from Amazon or a download site, take a moment to get on the mailing list for the book. I'll let you know when I release updates to the book.

- Get on the mailing list for the book

A Note to Reviewers and Teachers

This book approaches the subject differently than most introductions to Rails. In this book, rather than show the student how to use scaffolding, I introduce the model-view-controller design pattern by creating the components manually. Lastly, though every other Rails tutorial shows how to use a database, this book doesn't, because I want the book to be a short introduction and I believe the basic principles of a web application stand out more clearly without adding a database to the application. Though this tutorial is not a typical Rails introduction, I hope you'll agree that it does a good job in preparing Rails beginners for continued study, whether it is a course or more advanced books.

Using the Book in the Classroom

If you've organized a workshop, course, or code camp, and would like to assign the book as recommended reading, contact me at daniel@danielkehoe.com to arrange access to the book for your students at no charge.

Let's Get Started

In the next chapter, we'll learn how to get help when you need it.
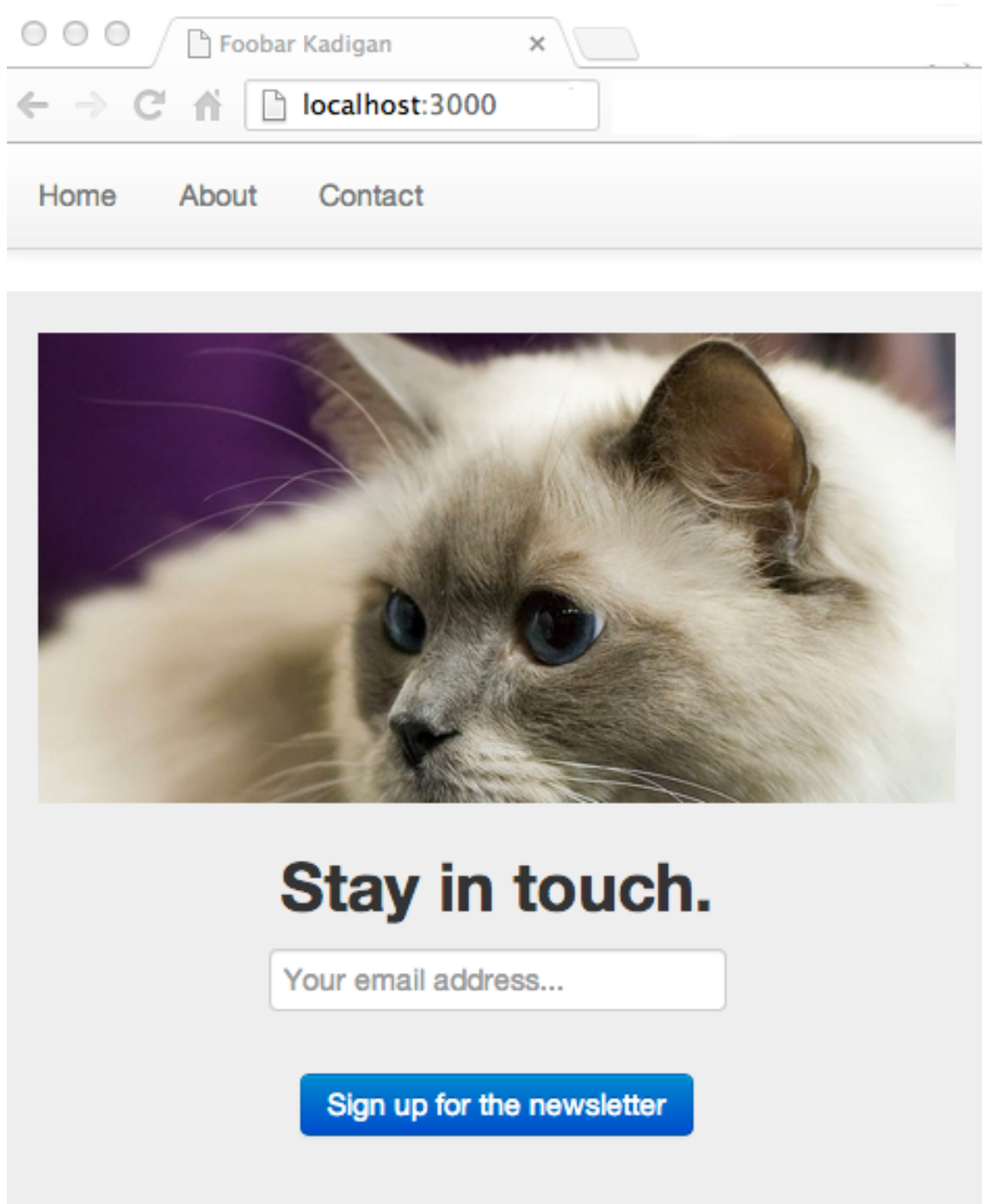
Figure 2.1: The application you will build.

Chapter 3

Get Help When You Need It

This chapter appears in the first book in this series, but I've included an expanded version here in Book Two because it is important.

Example Code

If you follow this tutorial closely, you'll have a working application that closely matches the example app in the learn-rails GitHub repository. If your application doesn't work after following the tutorial, compare the code to the example app in the GitHub repository, which is known to work.

Version Check

The code in this tutorial was tested by many people and worked flawlessly at the time this was written. The learn-rails example application on GitHub serves as a "reference implementation" if you have problems. The example application is updated more frequently than the published tutorial.

You'll find the version number and release date on the first page of this book

9

(under the book title). Check the learn-rails GitHub repository to find out if you have the newest version of the book. The README page on the GitHub repo always shows the most recent version number for the book and the tutorial application.

Don't worry if the learn-rails GitHub repository seems complicated or difficult to understand. You'll get to know GitHub over time and it will become familiar. For now, it is only important to know that it is there in case you have problems.

Take a moment now to look at the open issues on GitHub to see what problems you may encounter as you work your way through the tutorial. You can look at the closed issues to see some of the solved problems.

If you have trouble building the application in this book, and suspect something may be out of date, you can compare the Gemfile in the repo to see if we've changed gems or specified version numbers to avoid compatibility issues.

You can also check the CHANGELOG and look at recent commits to see the current state of the application.

Getting Help With the Book

Let's consider what to do if you encounter problems as you build the tutorial application in this book.

If you are in a classroom, or studying in a group, ask a peer to look at your problem. Most problems are caused by simple typos or formatting errors. Your classmate may see what you overlooked.

Stack Overflow provides a question-and-answer forum for readers of this book. As the author of this book, I can't solve your individual problems or help you directly by email. If I did, I would not have time to create the tutorials and videos that benefit so many people. However, I watch for questions on Stack Overflow. Everyone benefits when solutions are made public.

- tag your questions on Stack Overflow with **railsapps** for extra attention

If your questions aren't answered on Stack Overflow, try the Reddit forum:

- /r/rails

I sincerely hope you won't encounter obstacles as you build the tutorial application. Thousands of beginners have successfully completed the book and, unless a gem has recently changed, you should have no problem.

Now let's consider where to look for help when you are working on your own Rails projects.

Getting Help With Rails

Video Option

Watch the six minute video if you have subscribed:

- Get Help with Rails

Where To Go For Help?

What will you do when you get stuck?

"Google it," of course. But here's a trick to keep in mind. Google has options under "Search tools" to show only recent results from the past year. Use it to filter out stale advice that pertains only to older versions of Rails.

Stack Overflow is as important as Google for finding answers to programming problems. Stack Overflow answers are often included in Google search results, but you can go directly to Stack Overflow to search for answers to your questions. Like Google, answers from Stack Overflow are helpful if you check carefully to make sure the answers are recent. Also be sure to compare answers

to similar questions; the most popular answer is not always the correct answer to your particular problem.

Requests for advice (especially anything that provokes opinions) are often rejected on Stack Overflow. Instead, try Reddit for advice or recommendations. You'll find discussion forums ("subreddits") devoted to Rails and Ruby. You can also visit the Quora question-and-answer site for topics devoted to Rails and Ruby.

References

Here are suggestions for the most important references.

If you feel overwhelmed by all the links, remember that you can use this book to build the tutorial application without any additional resources. Right now, it's important to know additional help is available when you need it.

RailsGuides

The Rails Guides are Rails's official documentation, written for intermediate-level developers who already have experience writing web applications. The Rails Guides are an excellent reference if you want to check the correct syntax for Rails code. You'll be able to use the Rails Guides after completing this tutorial.

Cheatsheets

Tobias Pfeiffer has created a useful Rails Beginner Cheat Sheet that provides a good overview of Rails syntax and commands.

Even better than a cheatsheet, for Mac users, is an application named Dash that offers fingertip access to reference documentation for Ruby, Rails, HTML,

CSS, JavaScript, and many other languages and frameworks.

API Documentation

The API documentation for Ruby and Rails shows every class and method. These are extremely technical documents (the only thing more technical is reading the source code itself). The documents offer very little help for beginners, as each class and method is considered in isolation, but there are times when checking the API documentation is the only way to know for certain how something works.

- Rails Documentation - official API docs

- apidock.com/rails - Rails API docs with usage notes

- apidock.com/ruby - Ruby API docs with usage notes

I recommend Dash as a tool to look up classes, modules, and methods in Ruby and Rails. Dash is a macOS app; use Zeal on Linux. Dash and Zeal run offline (they don't need an Internet connection) so you can use them anywhere.

Meetups, Hack Nights, and Workshops

I'd like to urge you to find ways you can work with others who are learning Rails. Peer support is really important when you face a challenge and want to overcome obstacles.

Most large urban areas have meetups or user group meetings for Rails developers. Try Meetup.com or google "ruby rails (my city)". The community of Rails developers is friendly and eager to help beginners. If you are near a Rails meetup, it is really worthwhile to connect to other developers for help

and support. You may find a group that meets weekly for beginners who study together.

Local user groups often sponsor hack nights or hackathons which can be evening or weekend collaborative coding sessions. You don't have to be an expert. Beginners are welcome. You can bring your own project which can be as simple as completing a tutorial. You will likely find a study partner at your level or a mentor to help you learn.

If you are a woman learning Rails, look for one of the free workshops from RailsBridge or Rails Girls. These are not exclusively for women; everyone considered a "minority" in the tech professions is encouraged to participate; and men are included when invited by a woman colleague or friend.

Pair Programming

Learning to code is challenging, especially if you do it alone. Make it social and you'll learn faster and have more fun.

There's a popular trend in the workplace for programmers to work side-by-side on the same code, sharing a keyboard and screen. It's effective, both to increase productivity and to share knowledge, and many coders love it. When programmers are not in the same office, they share a screen remotely and communicate with video chat.

Look for opportunities to pair program. It's the best way to learn to code, even if your pairing partner is only another beginner. Learn more about pair programming on the site pairprogramwith.me and find a pairing partner at codermatch.me or letspair.net.

Remote pair programming requires tools for screen sharing and video chat. Pairing sessions often use:

- Google+ Hangouts

- Screenhero

- Floobits

- Cloud9 IDE

More tools are emerging as remote pair programming becomes popular.

Pairing With a Mentor

By far, the best way to learn is to have a mentor at your side as you undertake a project. That is an opportunity that is seldom available, unless you've been hired to be part of a team in a company that encourages pair programming.

You can try RailsMentors, a network of volunteer mentors offering free help.

If you can pay for help, find a mentor using HackHands or AirPair. Market rates are expensive for a student, obviously, but if you are learning on the job or building an application for your own business, connecting online with a mentor might be a godsend.

AirPair connects developers for real-time help using video chat and screen sharing. Experts set their own rate and the site matches you according to your budget. Expect to pay market rates for consulting ranging from USD $40 per hour to $150 per hour or more.

HackHands promises to instantly connect you with a qualified expert at a cost of one dollar per minute for mentorship using video chat and screen sharing.

Code Review

Code review is an essential part of the development process. There's always more than one way to implement a feature, and some ways are better than others, but you may not know it unless you ask someone to look at your code. When you pair with a mentor, you get the benefit of code review. But even

if you don't have a mentor, you can get code review online. StackExchange, the parent of StackOverflow, has a free site for code review, and a new service promises code review as a service:

- codereview.stackexchange.com

Expert code review will accelerate your learning faster than anything else.

Knowing where to go for help is important; it is just as important to stay current.

Staying Up-to-Date

Rails changes frequently and its community is very active. Changes to Rails, expert blog articles, and new gems can impact your projects, even if you don't work full-time as a Rails developer. Consequently, I urge you to stay up-to-date with news from the community.

I suggest signing up for two weekly email newsletters:

- Ruby Weekly

- Green Ruby News

Another weekly email newsletter is more technical, and focused on code arriving in the next version of Rails:

- This Week in Rails

For daily news about Rails, check Peter Cooper's RubyFlow site which lists new blog posts from Rails developers each day.

Also take a look at this list of top blogs for Rails developers:

- 45 Ruby Blogs

If you like podcasts, check out Ruby Rogues and Envy Labs's Ruby5.

Finally, you can follow @rails_apps on Twitter for news about the RailsApps project.

Chapter 4

Accounts You May Need

You'll need accounts with four services before you begin building the application in this book.

This tutorial will show you how to save your work using GitHub. You can sign up for a GitHub account for free. Every experienced Rails developer has a GitHub account; it is where we collaborate on our code, both commercial and open source projects.

We'll create a form that allows website visitors to "opt-in" to a mailing list. You'll need a MailChimp account, which is free. If you send email to a mailing list, you will find it is useful to have an account with MailChimp or a similar service.

We'll also send email from the application, which will require a SendGrid account. SendGrid is a transactional email service that lets you send email messages efficiently and reliably. SendGrid offers a free account that allows you to send 12,000 messages per month for free.

Finally, we'll deploy the tutorial application to Heroku which provides Rails application hosting. It costs nothing to set up a Heroku account and deploy as many applications as you want. It is the easiest way to deploy a Rails application and most Rails developers use Heroku at some time in their careers.

GitHub

Rails developers use GitHub for collaboration and remote backup of projects.

For this tutorial, I suggest you get a free personal GitHub account if you don't already have one. As a developer, your GitHub account establishes your reputation in the open source community. If you're seeking a job as a developer, employers will look at your GitHub account. When you work with other developers, they may check to see what you've worked on recently. Don't be reluctant to set up a GitHub account, even if you're a beginner. It shows you are serious about learning Rails.

You'll be asked to provide a username. This can be a nickname or short version of your real name (for example, your Twitter username).

You'll be asked to provide an email address. It's very important that you use the same email address for your GitHub account that you use to configure Git locally (there will be more about configuring Git later). If you create a Heroku account to deploy and host your Rails applications, you should use the same email address.

After you create your GitHub account, log in and look for the button "Edit Your Profile." Take a few minutes to add some public information to your account. It is really important to provide your real name and a public email address. Displaying your real name on your GitHub account makes it easy for people to associate you with your work when they meet you in real life, for example at a meetup, a hackathon, or a conference. Providing a public email address makes it possible for other developers to reach you if you ask questions or submit issues. If you can, provide a website address (even just your Twitter or Facebook page). In general, you won't be exposed to stalkers or spammers (except some recruiters) if you are open about yourself on GitHub.

Later I'll show you how to set up and use Git and GitHub.

MailChimp

This tutorial shows how website visitors can sign up to receive a newsletter provided by a MailChimp mailing list. MailChimp allows you to send up to 12,000 emails/month to a list of 2000 or fewer subscribers for free. There is no cost to set up an account.

MailChimp will ask you to provide a website address and company details for your account. These details are included when email messages are sent to your subscribers. If you don't have your own website, you can enter the URL for your GitHub account for now. Use your own name for a company if you don't have one.

After you have set up a MailChimp account, create a new mailing list where you can collect email addresses of visitors who have asked to subscribe to a newsletter. The MailChimp "Lists" page has a button for "Create List." The list name and other details are up to you.

If you get frustrated with the complex and confusing MailChimp interface, try to remember that the friendly MailChimp monkey is laughing with you, not at you.

SendGrid

Earlier editions of this book showed how to use a Gmail account to send email from the application. Google has taken steps to make Gmail more secure and now it can be difficult to send email from a Rails application using Gmail.

This tutorial provides instructions for SendGrid. SendGrid offers a free account that allows you to send 12,000 messages per month for free.

Scroll to the bottom of the SendGrid pricing page to see details about the free plan. Click the "Try for Free" link to set up an account. No credit card is needed.

Heroku

We'll use Heroku to host the tutorial application so anyone can reach it.

To deploy an app to Heroku, you must have a Heroku account. Visit https://signup.heroku.co
to set up an account.

Be sure to use the same email address you used to register for GitHub. It's
very important that you use the same email address for GitHub and Heroku
accounts.

Chapter 5

Get Started

Before you can start building, you'll need to install Ruby (the language) and Rails (the gem). I'll provide links to installation instructions that are up to date. Even if you've already installed Rails, please review the instructions to make sure your development environment is set up correctly. Other books and tutorials often skip important details.

Text Editor and Terminal Applications

I've explained how to use a text editor and terminal application in Book One. If you haven't used the Unix command line before, refer to Book One for an introduction.

I recommend the Atom text editor but you may use Sublime Text or any others that provide syntax highlighting. Refer to Book One for important instructions about setting up the text editor so you can open a file from the command line.

Copying and Pasting Code

In the next chapter you'll begin building a Rails application.

You need to get the code from this tutorial into your computer. You could just read and imagine, but really, building a working application is the only way to learn.

The most obvious way is to copy and paste from this tutorial into your text editor, assuming you are reading this on your computer (not a tablet or printed pages). It's a bit tedious and error-prone but you'll have a good opportunity to examine the code closely.

Some students like to type in the code, character by character. If you have patience, it's a worthwhile approach because you'll become more familiar with the code than by copying and pasting.

Don't feel shy about copying code; it's how you will learn. Working programmers spend a lot of time copying code from others. At first, you will copy a lot of code. As you gain proficiency, you will copy code and adapt it, more extensively as you gain confidence and skill. Only when you've been working full-time as a coder for months or years will you find yourself writing code from scratch; even then, when you encounter new problems, you will still look for code examples to copy and adapt.

Box 5.1. Warning About the PDF and Kindle Versions

This book is available in several formats, including online (HTML), PDF, ePub (Apple iBooks), and mobi (Kindle) versions.

Use the online edition of the book if you can. With the online edition, you'll be able to copy and paste the code without any problem. The ePub version (using Apple iBooks) also preserves line breaks and indentation when copying code.

Copying without line breaks will cause code errors. You'll lose line breaks when copying code with the following versions:

- PDF version on macOS using the Preview application

- mobi (Kindle)

If you use Adobe Acrobat you'll be able to copy the line breaks (though indenting is lost). You can also open a PDF file in Chrome or Safari web browsers and copy code with line breaks. With the mobi (Kindle) version, you'll have to carefully reformat the code after pasting into your text editor.

Indentation makes code more readable, so try to preserve the indentation you see in the code samples. In YAML files (with the file extension **.yml**), indentation is required (your application will break without it).

Your Computer

You can develop web applications with Rails on computers running Mac OS X, Linux, or Microsoft Windows operating systems. Most Rails developers use macOS or Linux because the underlying Unix operating system has long been the basis for open source programming.

Later in this chapter, I'll give links to installation instructions for macOS and Linux.

For Windows users, I have to say, installing Rails on Windows is frustrating and painful. Readers and workshop students often tell me that they've given up on learning Rails because installation of Ruby on Windows is difficult and introduces bugs or creates configuration issues. Even when you succeed in getting Rails to run on Windows, you will encounter gems you cannot install. For these reasons, I urge you to use Cloud9, a browser-based development environment, on your Windows laptop.

Hosted Computing

If you are using Windows, or have difficulty installing Ruby on your computer, try using Cloud9.

Cloud9 provides a hosted development environment. That means you set up an account and then access a remote computer from your web browser. The Cloud9 service is free for ordinary use. There is no credit card required to set up an account. You'll only be charged if you add extra computer memory or disk space (which you don't need for ordinary Rails development).

The Cloud9 service gives you everything you need for Rails development, including a Unix shell with Ruby pre-installed, plus a browser-based file manager and text editor. Any device that runs a web browser will give you access to Cloud9, including a tablet or smartphone, though you need a broadband connection, a sizable screen, and a keyboard to be productive.

Try the Terminal

Look for the Terminal application in the following places:

- MacOS: *Applications - Utilities - Terminal*

- Linux: *Applications - Accessories - Terminal*

- Windows: *Taskbar Start Button - Command Prompt*

On the Mac, search for the macOS Terminal application by pressing the Command-Spacebar combination (which Apple calls "Spotlight Search") and searching for "Terminal." The magnifying glass in the upper right corner of your screen will also launch "Spotlight Search." Or look in the **Applications/Utilities/** folder for the Terminal application. You'll need to click the name of the application to launch the Terminal.

For Linux or Windows, The Command Line Crash Course explains how to launch a terminal application.

Launch your terminal application now.

Try out the terminal application by entering a shell command.

```
$ whoami
```

Don't type the **$** character. The **$** character is a cue that you should enter a shell command. This is a longtime convention that indicates you should enter a command in the terminal application or console.

The Unix shell command **whoami** returns your username.

Don't type the **$** prompt.

You might see:

```
command not found: $
```

which indicates you typed the **$** character by mistake.

If you are new to programming, using a text editor and the shell will seem primitive compared to the complexity and sophistication of Microsoft Word or Photoshop. Software developers edit files with simple text editors and run programs in the shell. That's all we do. We have to remember the commands we need (or consult a cheatsheet) because there are no graphical menus or toolbars. Yet with nothing more than a text editor and the command line interface, programmers have created everything that you use on your computer.

Installing Ruby

Your first challenge in learning Rails is installing Ruby on your computer.

Frankly, this can be the most difficult step in learning Rails because no tutorial can sort out the specific configuration of your computer. Get over this hump and everything else becomes easy.

The focus of this book is learning Rails, not installing Ruby, so to keep the book short and readable, I'm going to give you links to articles that will help you install Ruby.

You'll spend at least an hour installing Ruby and Rails, so be sure to allow enough time for the task.

MacOS

See this article for macOS installation instructions:

Install Ruby on Rails - MacOS

Ubuntu Linux

See this article for Ubuntu installation instructions:

Install Ruby on Rails - Ubuntu

Hosted Computing

Cloud9 is a browser-based development environment. Cloud9 is free for small projects. If you have a fast broadband connection to the Internet, this is your best choice for developing Rails on Windows. And it is a good option if you have any trouble installing Ruby on Mac or Linux because the Cloud9 hosted environment provides everything you need, including a Unix shell with Ruby and RVM pre-installed, plus a browser-based file manager and text editor. Using a hosted development environment is unconventional but leading developers do so and it may be the wave of the future.

See this article for Cloud9 installation instructions:

Install Ruby on Rails - Cloud9

The article shows how to get started with Cloud9.

If you use Cloud9, be sure to pick the "Blank" template for your workspace, not the "Ruby" or "Ruby on Rails" templates that provide a prebuilt Rails application. We're building from scratch in this tutorial.

Windows

Here are your choices for Windows:

- Use the Cloud9 hosted development environment

- Install the Railsbridge Virtual Machine

- Use RubyInstaller for Windows

Cloud9 is ideal if you have a fast Internet connection. If not, download the Railsbridge Virtual Machine to create a virtual Linux computer using Vagrant. Other tutorials may suggest using RailsInstaller, but it will not provide an up-to-date version of Ruby or Rails. Also, RVM does not run on Windows.

Your Workspace

Take a moment to think about where on your computer you'll do your work and store your files. You may have a **documents/** folder. You could make a similar folder named **projects/** or **code/** or **workspace/** for your programming projects. Use the Unix `mkdir` command to create a folder or create it with your file browser.

If you haven't done so already, make a folder to contain your programming projects. You don't need to do this if you already created a **workspace/** folder in the Unix chapter in Book One.

```
$ cd ~
$ pwd
/Users/danielkehoe
$ mkdir workspace
$ cd workspace
```

In this tutorial, the terms "folders" and "directories" mean the same thing.

Use the Unix **cd** command to change directories.

When you enter the Unix command **cd ~**, you'll move to your home (or "user") directory. The squiggly **~** "tilde" character is a Unix shortcut that indicates your home folder.

The Unix **pwd** command shows the "present working directory," where you are.

The Unix **mkdir** command creates an empty folder and we move into it with the Unix **cd** command.

Video Option

Watch the four minute video if you have subscribed:

- [UNIX Commands Basics](#)

Understanding Version Numbers

Rails follows a convention named *semantic versioning*:

- The first number denotes a *major version* (Rails 4)

- The second number denotes a *minor release* (Rails 4.2)

- The third number denotes a *patch level* (Rails 4.2.1)

A major release includes new features, including changes which break backward compatibility. For example, switching from Rails 3.2 to Rails 4.0 required a significant rewrite of every Rails application.

A minor release introduces new features but doesn't break anything. For example, Rails 3.2 added the asset pipeline, and Rails 4.2 added the Active Job feature for background processing.

A patch release fixes bugs but doesn't introduce significant features. Usually this means you can change the version number in the Gemfile and run **bundle update** without making any other changes to your application.

Ruby and Rails Version Check

Check that appropriate versions of Ruby and Rails are installed in your development environment. You'll need:

- The Ruby language (version 2.3 or newer)

- The Rails gem (version 5.0 or newer)

Open your terminal application and enter:

```
$ ruby -v
```

You might see:

```
ruby 2.3.2p0 (...)
```

You've got Ruby version 2.3.2, patch level "p0" (Ruby versions add an extra patch level to semantic versioning). If you've got a newer version of Ruby, no problem; minor updates to Ruby don't affect Rails.

Try:

```
$ rails -v
```

You might see:

```
Rails 5.0.0
```

If you see:

```
Rails is not currently installed on this system.
```

You are not using an RVM gemset where the Rails gem is installed. Go to the Installing Rails instructions for your computer if you have not set up Rails. The next section explains more about RVM gemsets which may be all you need to find Rails.

If you have Rails 4.2 or older versions, you must update to Rails 5.0. See the Installing Rails instructions for your computer.

Versions such as `5.0.0.beta3` or `5.0.0.rc1` are beta versions or "release candidates." You can use a release candidate in the weeks before a final release becomes available.

If you've got Rails 5.0.1 or newer, that's fine. It means minor bugs have been fixed since this was written, but the book is still current. You can check for the current version of Rails here.

Box 5.2. Rails 5.0

This edition of the book was prepared using Rails 5.0.0.1. The newest version of the book is listed on the README page of the learn-rails GitHub repository. The online version of the book is always the newest version of the book.

RVM

I promised that this book would introduce you to the practices of professional Rails developers. One of the most important utilities you'll need in setting up a real-world Rails development environment is RVM, the Ruby Version Manager.

RVM lets you switch between different versions of Ruby. Right now, that might not seem important, but as soon as a new version of Ruby is released, you'll need to upgrade, and it is best to be ready by installing the current version of Ruby with RVM, so you can easily add a new version of Ruby later, and still switch back to older versions as needed.

RVM also helps you manage your collections of gems, by letting you create multiple *gemsets*. Each gemset is the collection of gems you need for a specific project. Rails changes frequently; with RVM, you can install a specific version of Rails in a project gemset, along with all the gems you need for the project. When a new version of Rails is released, you can create a new gemset with the new Rails version when you start a new project. Your old project will still have the version of Rails it needs in its own gemset.

If you've followed the instructions in the article Installing Rails and installed RVM, you'll be ready to handle multiple versions of Ruby, and multiple versions of Rails. That's as it should be. Most professional Rails developers have more than one version of Ruby or Rails, and RVM makes it easy to switch.

RVM will show you a list of available Ruby versions:

```
$ rvm list
```

You can see a list of available gemsets associated with the current Ruby version:

```
$ rvm gemset list
```

You will see an arrow that shows which gemset is active.

You will see a **global** gemset as well as any others you have created, such as a gemset for **Rails5.0**.

Here's how to switch between gemsets:

```
$ rvm gemset use global
```

And switch back to another:

```
$ rvm gemset use default
```

After you've worked on a few Rails applications, you'll see several project-specific gemsets if you are using RVM in the way most developers do.

RVM is not the only utility you can use to manage multiple Ruby versions. Some developers like Chruby or rbenv. Don't be worried if you hear debates about RVM versus Chruby or rbenv; developers love to compare the merits of their tools. RVM is popular, well-supported, and an excellent utility to help a developer install Ruby and manage gemsets; that's why we use it.

Project-Specific Gemset

For our learn-rails application, we'll create a project-specific gemset using RVM. We'll give the gemset the same name as our application.

By creating a gemset for our tutorial application, we'll isolate the current version of Rails and the gems we need for this project. Whether you use RVM or another Ruby version manager, this will introduce you to the idea of "sandboxing" (isolating) your development environment so you can avoid conflicts among projects.

After we create the project-specific gemset, we'll install the Rails gem into the gemset. Enter these commands:

```
$ rvm use ruby-2.3.2@learn-rails --create
$ gem install rails
```

This will install the newest version of Rails 5.0. It takes a few minutes to automatically install all the gems that are needed for Rails.

It's absolutely necessary to create a gemset and install Rails so we can move on to creating the application in the next chapter. If you have trouble at this point, refer to the article Installing Rails or the RVM website. Linux users may need to check instructions for Integrating RVM.

Let's make sure Rails is ready to run. Open a terminal and type:

```
$ rails -v
```

You should see the message "Rails 5.0.0" (or something similar).

Now let's explore the `rails new` command and get started building the tutorial application.

Chapter 6

Create the Application

In this chapter, we'll begin building the tutorial application. But first, let's consider the options you have for starter applications.

Starter Applications

Rails provides a *framework*; that is, a software library that provides utilities, conventions, and organizing principles to allow us to build complex web applications. Without a framework, we'd have to code everything from scratch. Rails gives us the basics we need for many websites.

Still, the framework doesn't give us all the features we need for many common types of websites. For example, we might want users to register for an account and log in to access the website ("user management and authentication"). We might want to restrict portions of our website to just administrators ("authorization"). We also might want to add gems that enhance Rails to aid development (gems for testing, for example) or improve the look and feel of our application (the Bootstrap or Foundation front-end frameworks). Developers often mix and match components to make a customized Rails stack.

Developers often use a *starter application* instead of assembling an application

from scratch. You might call this a "template" but we use that term to refer to the *view files* that combine HTML with Ruby code to generate web pages. Most experienced developers have one or more starter applications that save time when beginning a new project. The RailsApps project was launched to provide open source starter applications so developers could collaborate on their starter applications and avoid duplicated effort. After you gain some skill with this tutorial, you might use the RailsApps starter apps to instantly generate a Rails application with features like authentication, authorization, and an attractive design. At the end of this book, you'll learn about Rails Composer, a tool for building starter applications.

For now, we'll begin with the Rails default starter application.

Workspace Folder and RVM Gemset

Are you in the folder named **workspace/** you created earlier?

```
$ pwd
/Users/danielkehoe/workspace/
```

If you're not in your workspace folder, enter a Unix command to move to the folder:

```
$ cd ~/workspace
```

We already created a project-specific gemset using RVM. Make sure it's ready to use:

```
$ rvm use ruby-2.3.2@learn-rails
$ rvm gemset list
gemsets for ruby-2.3.2...
   (default)
   global
=> learn-rails
```

You should see an arrow pointing to the **learn-rails** gemset. If not, go back to the previous "Get Started" chapter.

Use "Rails New" to Build the Application

Let's go! We have selected a gemset, we have Rails installed, and we're in our **workspace/** folder. Let's build a Rails application!

To create the Rails default starter application, type:

```
$ rails new learn-rails
```

This will create a new Rails application named "learn-rails."

It takes a few minutes when the build script runs **bundle install**. Don't worry; just give it enough time to finish (but no more than five minutes even if your Internet connection is very slow).

In the future, you can give your application a different name. For this tutorial, it is VERY IMPORTANT that you use the name "learn-rails." You'll be copying code that assumes the name is "learn-rails;" it will save you trouble to use this name.

The **rails new** command will create ten folders and 93 files.

It will install 62 gems into your gemset.

After you create the application, switch to its folder to continue work directly in the application:

```
$ cd learn-rails
```

This is your project directory. It is also called the application root directory. You'll spend all your time inside this folder.

Type the `ls` command to show the folders and files in a directory. Soon we'll learn more about each of these folders and files.

```
$ ls
Gemfile       Rakefile      config        lib           test
Gemfile.lock  app           config.ru     log           tmp
README.md     bin           db            public        vendor
```

Make a Sticky Gemset

RVM gives us a convenient technique to make sure we are always using the correct gemset when we enter the project directory. It will create hidden files to designate the correct Ruby version and project-specific gemset. Enter this command to create the hidden files:

```
$ rvm use ruby-2.3.2@learn-rails --ruby-version
```

The `--ruby-version` argument creates two files, **.ruby-version** and **.ruby-gemset**, that set RVM every time we `cd` to the project directory. Without these two hidden files, you'd need to remember to enter `rvm use ruby-2.3.2@learn-rails` every time you start work on your project after closing the console.

If you see "ERROR: Gemset 'learn-rails' does not exist", perhaps you over-looked an earlier step in the *Project-Specific Gemset* section (in the previous chapter) where we created the learn-rails gemset.

After creating the two hidden files, check if they are there:

```
$ ls -1pa
./
../
.gitignore
.ruby-gemset
.ruby-version
Gemfile
```

```
Gemfile.lock
README.md
Rakefile
app/
bin/
config/
config.ru
db/
lib/
log/
public/
test/
tmp/
vendor/
```

The "a" flag in the Unix `ls -1pa` command displays hidden files. Each hidden file is listed with a dot (period or full stop) at the beginning of the filename. You'll notice `.ruby-gemset` and `.ruby-version`.

You'll also see two "special files" which are not files at all:

- `./` - an alias that represents the current directory

- `../` - an alias that represents the parent directory

Box 6.1. Hidden Files in Cloud9

If you're using Cloud9, you must change preferences to see hidden files. In the window that contains the file list, there is a gear icon (dark in color and difficult to see). Clicking the gear option will give you options:

- Show Root File System

- Show Home in Favorites

- Show Hidden Files

You must select all three options to see the hidden files.
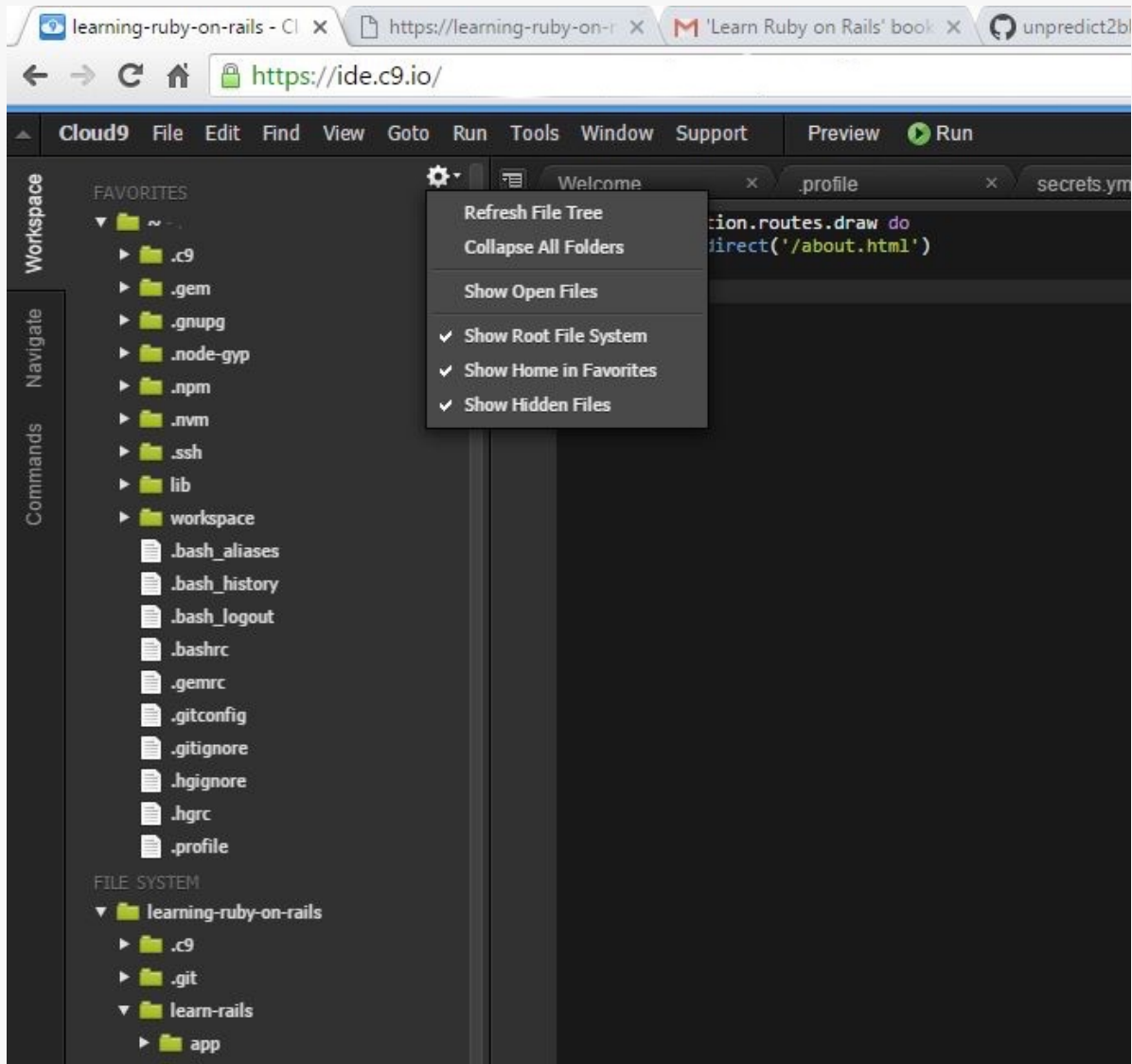
That's a brief diversion into Unix; let's try running our new Rails application.

Test the Application

You've created a simple default web application. It's ready to run.

Launching the Web Server

You can launch the application by entering the command:

```
$ rails server
```

Alternatively, to save typing, you can abbreviate the **rails server** command:

```
$ rails s
```

If you are using the Cloud9 hosted service, you'll need to enter **bin/rails server -p $PORT -b $IP**.

You'll see:

```
=> Booting Puma
=> Rails 5.0.0 application starting in development on http://localhost:3000
=> Run `rails server -h` for more startup options
=> Ctrl-C to shutdown server
Puma starting in single mode...
* Version 3.4.0 (ruby 2.3.2-p0), codename: Owl Bowl Brawl
* Min threads: 5, max threads: 5
* Environment: development
* Listening on tcp://localhost:3000
Use Ctrl-C to stop
```

The **rails server** command launches the Puma web server that is provided with Rails.

Errors for Linux Users

If you enter the command **`rails server`** and get an error message:

```
... Could not find a JavaScript runtime ...
```

You need to install Node.js. For help, see Install Ruby on Rails - Ubuntu.

Viewing in the Web Browser

To see your application in action, open a web browser window and navigate to
http://localhost:3000/. You'll see the Rails default information page.

Box 6.2. Viewing on a Hosted Platform

It is easy to see your web application in action on your local computer. If you are
using a hosted service such as Cloud9, it is a little more complicated.

If you are using **Cloud9**, click the "Preview" link in the IDE menu (at the top of
the page). There is a "Run" link, too, but it doesn't work if you have created your
Rails application in a folder within the **workspace/** folder. You can also open a
browser tab or window and enter the URL for the application, as hosted by Cloud9.
When you launch the Rails server, Cloud9 displays a helpful message showing the
URL where you can view your application.

Watch Log Messages

Notice that messages scroll in the console window when your browser requests
the Rails default web page.

Open the file **log/development.log** and you'll see the same messages. When a browser sends requests to the Puma web server, diagnostic messages are written to the console and to the **log/development.log** file. These diagnostic messages are an important tool for troubleshooting when you are developing.

Multiple Terminal Windows

You can keep more than one terminal window open. For convenience, you may want to keep a terminal window open for running the web server and watching diagnostic messages. In the Terminal or iTerm2 applications, Command-t opens additional console sessions in new "tabs."

Developers typically open more than one terminal window when they work on a Rails application. They'll start the server with the `rails server` command in one window (or tab) and watch the log messages. In another window (or tab), they'll enter commands as they build the application. They might create folders with a Unix command, run generators, or try out code with the `rails console` command (you'll learn about the `rails console` command in the "Troubleshoot" chapter).

To some people, the text editor and the terminal window look very similar. When you work on a file in a text editor, you make changes to one file, in one place. The terminal window is very different. Your computer can run multiple programs at once. You can open multiple terminal windows. In each terminal window, you can use the command line to launch a different program. Each program you start in a terminal window is a separate *process* and multiple processes can run simultaneously. You can end a process by pressing Control-c (in most cases), Control-d (in some cases), or closing the terminal window (almost always). From this perspective, a terminal window is a tool you use to launch processes and your computer is a machine that runs processes.

Stop the Web Server

You can stop the server with Control-c to return to the command prompt.

Most of the time you'll keep the web server running as you add or edit files in your project. Changes will automatically appear when you refresh the browser or request a new page. There is a tricky exception, however. If you make changes to the Gemfile, or changes to configuration files, the web server must be shut down and relaunched for changes to be activated.

As a rule of thumb, files that produce web pages can be changed without a restart. This includes any file in the **app/** folder which creates web pages, as well as the **config/routes.rb** file. Changes to files that create the environment for the web application, such as gems or configuration files, and are loaded at web server launch, won't be seen until the web server is restarted.

Get Organized for Efficiency

Before we learn about the Rails directory structure, take a minute to organize your screen real estate. During development, you'll jump between the console in a terminal application, your text editor, and a web browser window. As a Rails developer, you'll do this constantly, so think about how you can do this efficiently. Multiple screens make it easy, but even on a laptop you can get organized for efficiency.

Here's some ideas. Open a window in the terminal application, place it on the left side of your screen, and stretch it to the maximum vertical height of your screen. Open multiple tabs in your terminal application. Keep one tabbed window open for entering shell commands (like `cd` or `ls`) and another terminal window open for running the `rails server` command and viewing the log output.

Place your text editor window next to the terminal window and stretch it to full vertical height. If you are using Atom or Sublime Text, you can open two editor panels side-by-side. Some developers find it helpful to leave the file browser
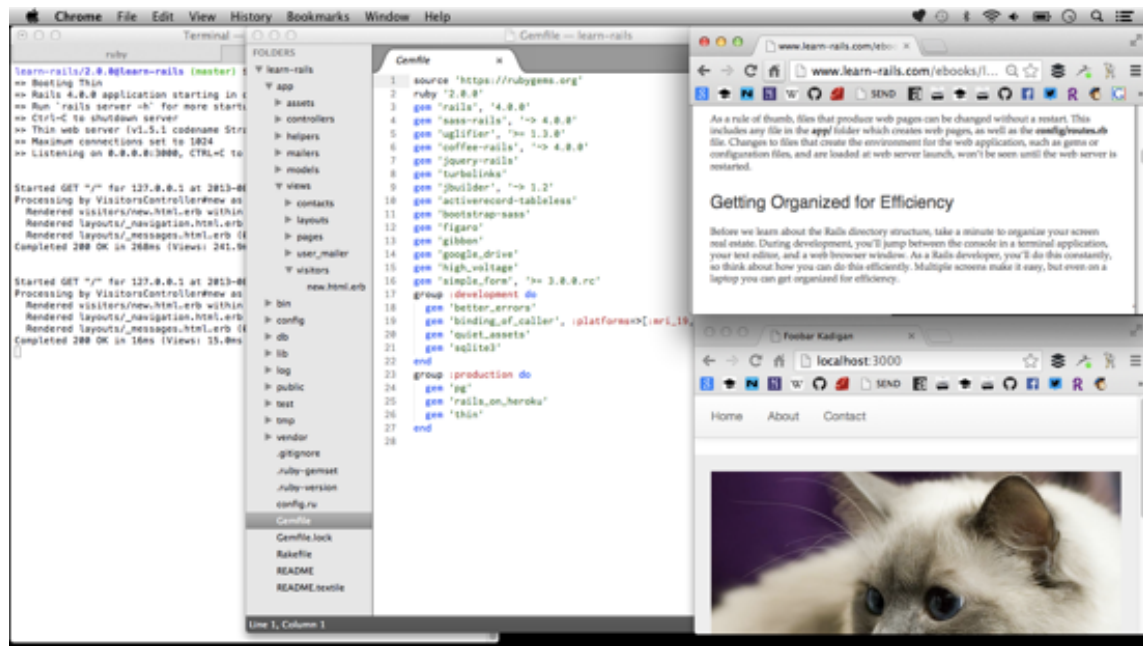
Figure 6.1: Getting organized for efficiency.

panel open to navigate the project directory; others hide the file browser panel to save space.

If you have enough screen space, leave your web browser open and place it next to your text editor. If your screen space is limited, you may have to overlap the web browser with the text editor, but position your web browser window so you can bring it to the front with a single click. You'll need multiple tabs open in your web browser. Unless you like constant distraction, close Gmail, Facebook, Twitter, and Hacker News. Open tabs for http://localhost:3000/, this tutorial, and additional references or documentation.

On the Mac, there are window management utilities that reposition windows with just a click or keyboard command; I use Moom but you can find others if you search for "mac window management utilities."

This is just a guide; I'm sure you can improve upon these suggestions.

Chapter 7

The Parking Structure

We've created the default Rails starter application.

The **`rails new`** command has created a project directory for us.

It is a parking structure for our code. Unlike an ordinary parking structure, where you park anywhere you like, this garage has assigned parking. You have to park your code in the right place. This is Rails, where convention brings order to the development process.

As you develop a web application, you'll do all your work in the project directory. It is important to know your way around and understand the purpose of each folder and file.

If you've built simple websites with HTML and CSS, or built websites with unstructured platforms such as Perl or PHP, you'll be surprised at the complexity of the Rails project directory. Rails is a software machine with many moving parts; the project directory provides a structure to manage the complexity. The logic and order of the project directory structure is familiar to every Rails developer, and consistent for every Rails application, which makes it easy to collaborate, maintain an application, and create open source projects.

Video Option

Watch the eleven minute video if you have subscribed:

- [Rails Project Directory](#)

Project Directory

Use the Unix **ls** command to list the contents of the project directory. For a one-column list that shows each subdirectory (marked with a slash), we'll add the **-1p** option to the command.

```
$ ls -1p
```

You'll see:

```
Gemfile
Gemfile.lock
README.md
Rakefile
app/
bin/
config/
config.ru
db/
lib/
log/
public/
tmp/
vendor/
```

Now is a good time to open a file browser window and look at the contents of the project directory. On the Mac, there's a command you can use to open the graphical file browser from the console. If you're in the project directory, type **open .**. The period (or "dot") is a Unix symbol that means "the directory I'm in."

```
$ open .
```
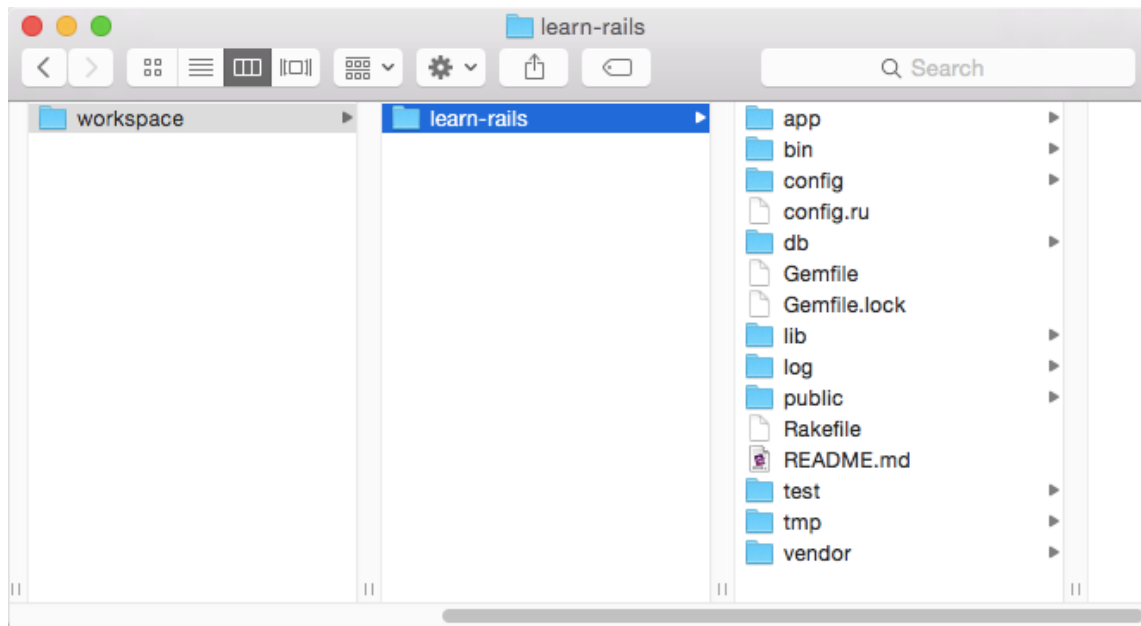

Figure 7.1: Rails directory structure.

You'll learn more about each file and folder as you proceed through the tutorial.

Get to Know the Folders and Files

To get you started, here are three tables. The first describes the files and folders that are important for every beginner. The second table describes the files and folders that you can ignore. The third table is a preview of things to come.

Important Folders and Files

These folders and files are **important** to beginners. This is where you will spend your time in Rails.

Gemfile	Lists all the gems used by the application.
Gemfile.lock	Lists gem versions and dependencies.
README.md	A page for documentation.
app/	Application folders and files.
config/	Configuration folders and files.
db/	Database folders and files.
public/	Files for web pages that do not contain Ruby code, such as error pages.

Not-So-Important Folders and Files

These folders and files are **not important** to beginners.

Rakefile	Scripts for the Rake utility program.
bin/	Folder for binary (executable) programs.
config.ru	Configuration file for Rack (a software library for web servers).
lib/	Folder for miscellaneous Ruby code.
log/	Folder for application server logfiles.
tmp/	Temporary files created when your application is running.
vendor/	Folder for Ruby software libraries that are not gems.

Folders for Testing

spec/	Folder for the popular RSpec testing framework.
test/	Folder for the default Rails testing framework.

The **test/** folder is present in the default Rails starter app. You'll use the **test/** folder when you learn about test-driven development. Many Rails developers

use a different gem for testing, named RSpec, and your RSpec tests will go in a **spec/** folder.

The App Directory

Take time to drill down into the **app/** folder in the project directory. This is easiest using the file browser.
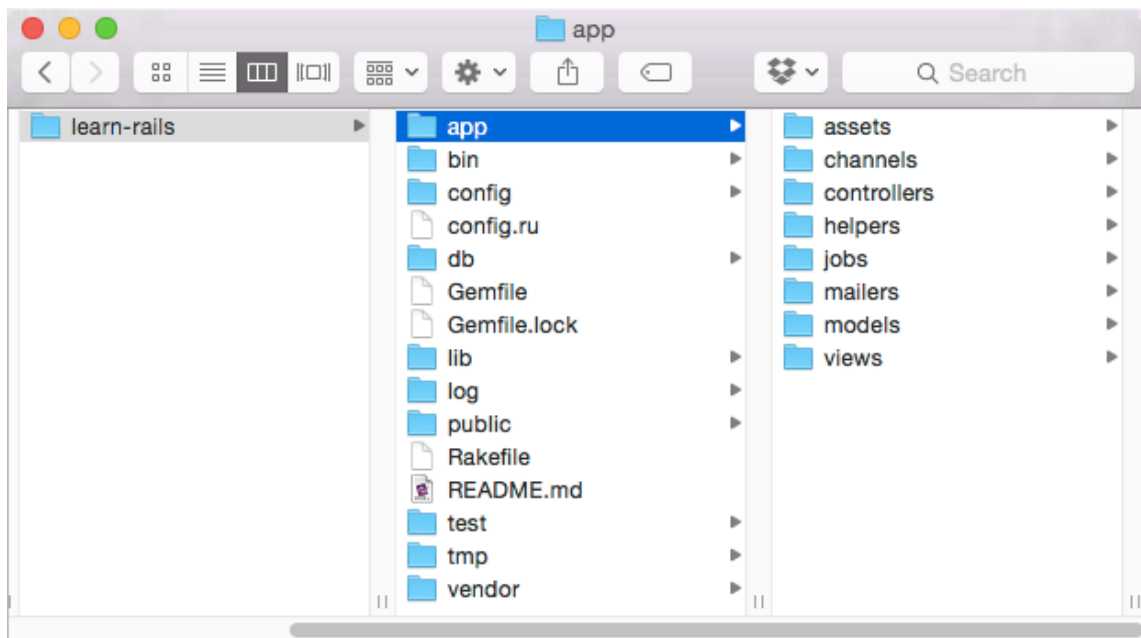
Figure 7.2: Rails app folder.

You can also use your text editor to view the folder.

Or do it with Unix commands:

```
$ cd app
$ ls -1p
assets/
channels/
controllers/
```

```
helpers/
jobs/
mailers/
models/
views/
```

Whether you use the file browser, Unix commands, or your text editor, you are looking at the same file system.

Most of the work of developing a Rails application happens in the **app/** folder.

Earlier we described Rails as "a set of files organized with a specific structure." We said the structure is the same for every Rails application. The **app/** directory is a good example. The folders in the **app/** directory are the same in every Rails application. This makes it easy to collaborate with other Rails developers, providing consistency and predictability.

- **assets**

- **channels**

- **controllers**

- **helpers**

- **jobs**

- **mailers**

- **models**

- **views**

You may recall our earlier description of Rails from the perspective of a software architect. In this folder, you'll see evidence of the model–view–controller design pattern. Three folders named **models/**, **views/**, and **controllers/** enforce the software architect's "separation of concerns" and impart structure to our

code. As you build the application, we'll explain the role of the MVC components in greater detail.

Five folders play supporting roles. The mailers folder is for code that sends email messages. The helpers folder is for Rails *view helpers*, snippets of reusable code that generate HTML. Later, when we learn more about *views*, we'll say view helpers are like "macros" that expand a short command into a longer string of HTML tags and content. Rails 3.1 added the **assets/** folder as a location for CSS and JavaScript files. The **jobs/** folder is for background jobs built with the Rails ActiveJob feature. Rails 5.0 added the **channels/** folder for the ActionCable feature which uses WebSockets for real-time communication between web server and browser.

Folders of Future Importance

You won't encounter these when you are a beginner:

policies/	Folder for code that controls access to features
services/	Folder for code that reduces the complexity of models and controllers

If you join a project to work on a large and complex Rails application, you may see folders such as these in the **app/** directory. As an application grows in complexity, an experienced software architect may suggest reducing the size of models and controllers by moving code to "POROs" (*plain old Ruby objects*). Code in any folder in the **app/** directory is shared throughout a Rails application without any additional configuration (in contrast, code you add to the **lib/** directory is only available with some extra work). Rails provides a basic model–view–controller framework but it is often necessary to extend it with code in a **services/** folder. Similarly, a **policies/** folder can be used to consolidate code that controls access to various features or pages of a web application.

Use the `cd ..` command ("change directory dot dot") to return to the project directory.

```
$ cd ..
$ pwd
/Users/danielkehoe/workspace/learn-rails
```

As a Rails developer, you'll spend most of your time navigating the hierarchy of folders as you create and edit files. And because Rails provides a consistent structure, you'll quickly find your way on any unfamiliar project.

Chapter 8

Time Travel with Git

Now that we've looked at our Rails project directory from the viewpoint of a programmer and software architect, let's consider the viewpoint of the time traveler.

This chapter will introduce you to software *source control*, also called *version control* or *revision control*. The terms all have the same meaning. At first sight, the concept seems rather dull, like sorting your socks. But it makes professional software development possible and, at the core, it is essentially a form of time travel.

To understand time travel, we need to understand *state*. It's a term you'll encounter often in software development. We know about states of matter. Water can be ice, liquid, or steam. Imagine a machine with a button that, each time it is pressed, changes water from one state to another. We call this a *state machine*. Almost every software program is a state machine. When a program receives an input, it transitions from one state to another. Like flipping a light switch, there's no in-between. Light or dark. Ice, liquid, or steam. Or, in a web application: logged in, logged out.

When we write software code, there's a lot of in-between. We look things up, we think, we type errors and we make corrections. As humans, we spend a lot of time in a flow of undetermined state. We can save our work at any time, but

we may be saving typos or unfinished code that doesn't work. Every so often, we get to a point where a task is finished; we've fixed all our errors and our code runs. We want to preserve the state of our work. That's when we need a version control system.

A version control system does more than a software application's "Save" command. Like a "Save" command, it preserves the current state of our files. It also allows us to add a short note that describes the work we've done. More importantly, it archives a snapshot of the current state in a *repository* where it can be retrieved if needed.

Here's where the time travel comes in. We can go back and recover the state of our work at any point where we committed a snapshot to the repository. In software development, travel to the past is essential because we often make mistakes or false starts and have to return to a point where we know things were working correctly.

What about time travel to the future? Often we need to try out code we may decide to discard, without disturbing work we've done earlier. Version control systems allow us to explore alternative futures by creating a *branch* for our work. If we like what we've done in our branch, we can merge it into the main trunk of our software project.

Unlike time travel in the movies, we can't travel back to any arbitrary point in the flow of time. We can only travel to past or future states we've marked as significant by checking our work into the repository.

Git

The dominant version control system among Rails developers is Git, created by the developer of the Linux operating system.

Unlike earlier version control systems, Git is ideal for wide-scale distributed open source software development. Combined with GitHub, the "social coding" website, Git makes it easy to share and merge code. When you work with

others on a project, your Git *commit messages* (the notes that accompany your snapshot) offer a narrative about the progress of the project. Well-written commit messages describe your work to co-workers or open source collaborators.

GitHub's support for *forking* (making your own copy of a repository) makes it possible to take someone else's project and modify it without impacting the original. That means you can customize an open source project for your own needs. You can also fix bugs or add a feature to an open source project and submit a *pull request* for the project maintainer to add your work to the original. Fixing bugs (large or small) and adding features to open source projects are how you build your reputation in the Rails community. Your GitHub account, which shows all your commits, both to public projects and your own projects, is more important than your resumé when a potential employer considers hiring you because it shows the real work you have done.

Collaboration is easy when you use a *branch* in Git. If you and a coworker are working on the same codebase, you can each make a branch before adding to the code or making changes. Git supports several kinds of *merges*, so you can integrate your branch with the trunk when your task is complete. If your changes collide with your coworker's changes, Git identifies the conflict so you can resolve the collision before completing the merge.

All the power of Git comes at a price. Git is difficult for a beginner to learn, largely because many of its procedures have no real-world analog. Have you noticed how time travel movies require mental gymnastics, especially when you try to make sense of alternative futures and intersecting timelines? Git is a lot like that, mostly because we use it to do things we don't ordinarily do in the real world.

In this tutorial, you won't encounter Git's advanced procedures, like resolving merges or reverting to earlier versions. We'll stick to the basics of archiving our work (and in one case, discarding work that we've done for practice). You can build the tutorial project without using Git. But I urge you to use Git and a GitHub account for this project, for two reasons. First, with your tutorial application on GitHub, you'll show potential employers or collaborators that you've successfully built a useful, functioning Rails application. More impor-

tantly, you must get to know Git if you plan to do any serious coding, either as a professional or a hobbyist.

Before I show you Git commands, I want to mention that some people use graphical client applications to manage Git. MacOS has GitHub for Mac, Git Tower, and other Mac Git clients. Graphical applications for Git are useful for colleagues who don't use a Terminal application, such as graphic designers or writers. There's no need for you to install these applications. Every developer I've met uses Git from the command line. It will take effort to master Git; the commands are not intuitive. But it is absolutely necessary to become familiar with Git basics.

Before you do any work on the tutorial application, I'll show you the basics of setting up and using Git.

Is Git Installed?

As a first step, make sure Git is installed on your computer:

```
$ which git
/usr/local/bin/git
$ git version
git version ...
```

If Git is not found, install Git. See the article Rails with Git and GitHub for installation instructions.

Is Git Configured?

Make sure Git knows who you are. Every time you update your Git repository with the `git commit` command, Git will identify you as the author of the changes.

```
$ git config --get user.name
$ git config --get user.email
```

You should see your name and email address. If not, configure Git:

```
$ git config --global user.name "Real Name"
$ git config --global user.email "me@example.com"
```

Use your real name so people will associate you with your work when they meet you in real life. There's no reason to use a clever name unless you have something to hide. And use your full name, not just your first name.

Use the same email address for Git, your GitHub account, and Heroku to avoid headaches.

Create a Repository

Now we'll add a Git repository to our project. It's a basic step you'll repeat every time you create a new Rails project.

Extending the time traveler analogy, initializing a Git repository is equivalent to setting up the time machine.

Be sure you are in your project directory, not your user home directory or somewhere else. Use the **pwd** command to check:

```
$ pwd
/Users/danielkehoe/workspace/learn-rails
```

The **git init** command sets up a Git repository (a "repo") in the project directory. We add the Unix symbol that indicates Git should be initialized in the current directory (git init dot):

```
$ git init .
Initialized empty Git repository in ...
```

It creates a hidden folder named **.git/** in the project directory. You can peek at
the contents:

```
$ ls -1p .git
HEAD
config
description
hooks/
info/
objects/
refs/
```

All Git commands operate on the hidden files. The hidden files record the
changing state of your project files each time you run the **git commit** com-
mand. There is no reason to ever edit files inside the hidden **.git/** folder (doing
so could break your time machine).

GitIgnore

The hidden **.git/** folder contains the Git repository with all the snapshots of
your changing project. The snapshots are highly compressed, only containing
records of changes, so the repository takes up very little file space relative to
the project as a whole.

Not every file should be included in a Git snapshot. Here are some types of
files that should be ignored:

- log files created by the web server

- database files

- configuration files that include passwords or API keys

Git gives us an easy way to ignore files. A hidden file in the project directory named **.gitignore** can specify a list of files that are never seen by Git. The `rails new` command creates a **.gitignore** file with defaults that include log files and database files. Later, when we add configuration files that include secrets, we'll update the **.gitignore** file.

Take a look at the contents of the **.gitignore** file. We use the Unix `cat` command to display the contents of the file:

```
$ cat .gitignore
# See https://help.github.com/articles/ignoring-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
/db/*.sqlite3-journal

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

# Ignore Byebug command history file.
.byebug_history
```

For a **.gitignore** file that ignores more, see an example .gitignore file from the RailsApps project.

Git Workflow

Your workflow with Git will move through four distinct phases as you add or edit files.

Untracked Files

The first phase is a "dirty" state of untracked and changed files, before any snapshot. The **git status** command lists all folders or files that are not checked into the repository.

```
$ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .gitignore
        Gemfile
        Gemfile.lock
        README.md
        Rakefile
        app/
        bin/
        config.ru
        config/
        db/
        lib/
        log/
        public/
        test/
        tmp/
        vendor/

nothing added to commit but untracked files present (use "git add" to track)
```

In the online version of this book (the HTML version), the example above appears in green. In the terminal window on your computer, it will be red, showing you untracked files.

Here the `git status` command tells us that we have many untracked files. We have created new files and they are saved on the computer's hard disk but nothing has been recorded in the Git repository.

Staging

I call this step, "Pose for your snapshot."

Recording files in the Git repository takes two steps: staging and committing. There will be times when you change many files at once. For example, you may fix a bug, add a new graphic, and change a form. You might think you'd like to have Git automatically record all the changes as you save each file. But the story of your project would be confusing and overly detailed. Git requires you to mark one or more files ("staging") before recording the changes ("committing"). This gives you fine-grained control over the recorded history of your project.

You can mark individual files to be staged:

```
$ git add Gemfile
```

Adding individual files allows you to selectively record the history of your project. For example, you might stage and commit a series of bug fixes before you stage and commit new features. Applying the time traveler analogy, it will be easier to travel back to look at bug fixes if they are not mixed in with new features.

More often, you'll mark all the files to be staged. Do so now:

```
$ git add -A
```

Running `git status` will show you a long list of files that are staged and ready to commit.

There are three forms of the `git add` command:

- `git add foo.txt` adds a file named **foo.txt**

- `git add .` adds all new files and changed files, except deleted files

- `git add -A` adds everything, including deletions

If it seems nonsensical that the command `git add -A` "adds deletions," don't worry. Like time travel, Git will stretch your understanding of what makes sense.

Most often, you can simply use the `git add -A` form of the command.

Now that you've marked the files that will be committed to the repository, you've told everyone to pose, and you're ready to take the snapshot.

Committing

The previous step, the "posing" step, or staging, gives you an opportunity to select particular files before you commit. If you've only worked on one feature, you'll likely stage and commit all your files.

The next step is a "commit" which I like to call, "clicking the snapshot."

When you "make a commit", you include a message that describes the work you've done. For a time traveler, the "commit message" is important; you are leaving a trail to help you find your way into the past. Google will show you dozens of blog posts about "writing better commit messages" but common sense can be your guide. For example, writing "fix registration form to catch blank email addresses" will be more helpful than merely writing "fix bugs." And if you wonder why commit messages are commonly written in the imperative not past tense ("fix" not "fixed"), it's a time traveler convention.

Now commit your project to the repository:

```
$ git commit -m "Initial commit"
```

The **-m** flag lets you add a message for the commit.

The pristine state of your new Rails application is now recorded in the repository on your local computer.

Running **git status** will tell you "nothing to commit, working directory clean."

```
$ git status
On branch master
nothing to commit, working directory clean
```

You've recorded your snapshot locally. Next let's see a list of previous snapshots. Then we'll learn how to save your snapshots remotely to GitHub.

Git Log

You can use the **git log** command to see your project history:

```
$ git log
commit 8da41eec9e864ed91b4a445d8cefdf7893e2faf6
Author: Daniel Kehoe <daniel@danielkehoe.com>
Date:   Fri Dec 18 10:30:12 2015 +0700

    Initial commit
```

The long string of characters that follows "commit" is an ID, or marker, that will help you travel back in time if you need to do so.

If you get "stuck" in **git log**, type **q** to return to the command prompt.

I like to use the **git log** command with an option for a compact listing:

```
$ git log --oneline
8da41ee Initial commit
```

The listing is easier to review when it is displayed in a compact format. The commit ID is abbreviated but it is all you need to travel back in time.

Repositories

When we talk about repositories, or "repos," we mean the archive of our git commits. The local repo is located in the hidden folder named **.git/** in the project directory. Of course, if your hard drive crashes, or your computer is lost or stolen, you'll lose your local repo along with your project. So it is wise to save your repository in the cloud. The GitHub site is a place to save repositories. GitHub is also a place for collaboration. Most Rails developers save their repositories to Github, as either a public repo for open source projects, or a private repo for proprietary projects.

Pushing to GitHub

We've seen three phases of the Git workflow: *untracked*, *staged*, and *committed*.

A fourth stage is important when you work with others: *pushing* to GitHub.

The repositories hosted on your GitHub account establish your reputation as a Rails developer for employers and developers you may work with. Even if your first project is copied from a tutorial, it shows you are serious about learning Rails and studying conscientiously.

Did you create a GitHub account? Now would be a good time to add your repo to GitHub.

Go to GitHub and create a new empty repository for your project. Name the repository "learn-rails" and give it a description. If the repository is public,

hosting on GitHub is free. Don't be reluctant to go public with an unfinished or half-baked project; everyone expects projects on GitHub to be works in progress.

Add GitHub as a remote repository for your project and push your local project to GitHub. Before you copy and paste the command, notice that you need to insert your own GitHub account name. In other words, change **YOUR_GITHUB_ACCOUNT** in the command shown below (or face problems described here).

```
$ git remote add origin https://github.com/YOUR_GITHUB_ACCOUNT/learn-rails.git
$ git push -u origin master
```

The **-u** option sets up Git so you can use **git push** in the future without explicitly specifying GitHub as the destination.

Now you can view your project repository on GitHub at:

- https://github.com/YOUR_GITHUB_ACCOUNT/learn-rails

Obviously you must change **YOUR_GITHUB_ACCOUNT** in the web address to see your own repository.

Take a look. It's an exact copy of the project on your local computer.

If you haven't used GitHub before, take some time to explore. GitHub is absolutely essential to all open source Rails development.

You may notice that the **README.md** file is automatically incorporated into the home page of the project repository on GitHub. For our next step, we'll update the README file, commit it to the local repo, and push it up to GitHub. It will be good practice for using Git.

The README

Changing the README file is a good way to practice with Git. It's also a good habit to edit the README file whenever you create a new project. It's easy to

neglect the README for little projects that you've just started. But replacing a default README file shows you are a disciplined, conscientious developer who will be a good collaborator.

The new README file can be brief. Just state your intentions and acknowledge any code you've borrowed. For this project you could say, "Excited to learn Rails with help from Daniel Kehoe's book!"

In your text editor, open the file **README.md** and replace the contents:

```
# Learning Rails

Learning Rails with a tutorial from [learn-rails.com](http://learn-rails.com/).
```

GitHub lets you add formatting using your choice of markup syntax, depending on the file extension you add to the filename:

- README.md uses the GitHub Flavored Markdown syntax

- README.textile uses the Textile syntax

- README.rdoc uses the rdoc syntax

We'll use Markdown syntax by adding the **#** character before the first line of text to force a headline. And we'll add a link that leads to the learn-rails.com website.

There's no requirement that you use Markdown syntax in your README file. Markdown simply is a popular way to add formatting to improve readability.

Use **git status** to see what has changed:

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
```

```
  modified:    README.md

no changes added to commit (use "git add" and/or "git commit -a")
```

Here's our typical workflow. We'll stage and commit the change:

```
$ git add -A
$ git commit -m "update README"
```

Then we'll push the change to GitHub:

```
$ git push origin master
```

If you decide not to use GitHub for this tutorial, you can skip this step (and skip it throughout the tutorial).

Take a look at your GitHub repository (refresh the web page). Very cool! The README file has been updated.

The `git log` command will display your project history:

```
$ git log --oneline
69b9b6c update README
8da41ee Initial commit
```

To learn more about Git, I recommend the book Learn Enough Git to Be Dangerous by my colleague Michael Hartl. An online version is available for free.

Now that you're comfortable with Git, we can begin customizing our new Rails application.

Chapter 9

Gems

The art of selecting gems is at the heart of Rails development. I explained earlier that gems are packages of code, "software libraries," that have been developed and tested by other developers. Some gems add functionality or features to a website. Other gems play a supporting role, making development easier or implementing basic infrastructure. Gems are open source. They are available at no charge and can be freely copied and modified.

Videos

If you have subscribed, two short videos introduce gems:

- [What Are Rubygems](#)

- [Find Rubygems](#)

RubyGems

It is a mark of honor to release a gem for public use, and a developer's reputation can be established when a gem becomes popular and widely used. Gems are often created when a developer has used the same code as a component in more than one web application. He or she will take time to release the code as a gem. That's how the Rails ecosystem was built, gem by gem since 2004.

There is no evaluation or review process in publishing gems. Gems are hosted on a public server, rubygems.org. Gems are mostly text files (like any other Ruby code), organized in a particular format with some descriptive information (in a **gemspec** file), and compressed and archived as a single file. A single command, `gem push`, uploads a gem to the rubygems.org server for anyone to use.

Over 50,000 gems have been released since rubygems.org was established. Some of these gems are used by one or two developers on their own projects. Many others have been neglected and abandoned due to lack of interest. Only a few thousand gems are popular and widely used. As a Rails developer, you must master the art of finding and evaluating gems so you can base your applications on the tried-and-true work of others.

There is no single authoritative source of recommendations for gems. The Ruby Toolbox website categorizes and ranks many gems by popularity, and it is a good place to begin hunting for useful gems. Other than that, it is useful to study example applications and search for blog posts to find which gems are most often recommended. When you find an interesting gem, search Stack Overflow or Google to see what people are saying. Look at the gem's GitHub repository and check:

- How many issues are open? How many are closed?

- How recent are the commits of patches or updates?

- Is there a CHANGELOG file?

- Is the gem well-documented?

- How many "stars" (people favoriting) or "forks" (people hacking)?

Popular gems are likely to have many reported issues, some of which are trivial problems or feature requests. Gems that are actively maintained will have many closed issues and, ideally, only a few open issues. When you find a gem that has many open issues and no recently closed issues, you've probably found a gem that has been abandoned. Also look at the commit log, which you'll find on the GitHub project page in a tab at the top of the page. Regular and recent activity in the commit log indicates the gem is actively maintained.

Rails Gems

Rails itself is a gem that, in turn, requires a collection of other gems. This becomes clear if you look at the summary page for Rails on the rubygems.org site. On that page, you'll see photos of the Rails core team. More importantly, you'll see a list of gems that are required to use Rails:

- actioncable - real-time communication using WebSockets

- actionmailer - framework for email delivery and testing

- actionpack - framework for routing and responding to web requests

- actionview - view templates and rendering

- activejob - queueing slow tasks to run in the background

- activemodel - architecture for model objects

- activerecord - framework for connections to databases

- activesupport - utility classes and Ruby library extensions

- bundler - utility to manage gems

- railties - console commands and generators

- sprockets-rails - support for the Rails asset pipeline

These are the "runtime dependencies" for Rails. Each of these gems has its own dependencies as well. When you install Rails, a total of 63 gems are automatically installed in your development environment.

Gems for a Rails Default Application

In addition to the Rails gem and its dependencies, a handful of other gems are included in every `rails new` default starter application:

- sqlite3 - adapter for the SQLite database

- puma - web application server

- sass-rails - enables use of the SCSS syntax for stylesheets

- uglifier - JavaScript compressor

- coffee-rails - enables use of the CoffeeScript syntax for JavaScript

- jquery-rails - adds the jQuery JavaScript library

- turbolinks - faster loading of webpages

- jbuilder - utility for encoding JSON data

You may not need a SQLite database, SCSS for stylesheets, jQuery or the others, but many developers use these tools so they are included in the default starter application.

Where Do Gems Live?

Gems are files saved in the computer's disk storage, containing someone else's code that you can use in your own application.

When you run a Rails application, gems are loaded into the computer's working memory immediately before your own custom code is loaded. Gems are handled by the Ruby interpreter no differently than your own code. It's all Ruby code, whether you or someone else wrote it. When you are building an application in Rails, you don't need to think about where gems are stored in your file system. It's all handled automatically.

Experienced programmers who have used software libraries in other languages might wonder how it works. Here's the technical explanation from the experts. Ruby has a **require** method that allows you to import software libraries into your programs. RubyGems extends the **require** method, adding gem directories to a **$LOAD_PATH**. When Rails loads, it will automatically require each of the gems listed in your Gemfile, finding the gems in the **$LOAD_PATH** directories.

If you're a curious person, you might like to see where the gems live. You can run the **gem env** command to reveal the RubyGems environment details which are normally hidden from you:

```
$ gem env
RubyGems Environment:
  - RUBYGEMS VERSION: 2.6.4
  - RUBY VERSION: 2.3.2 (2016-04-26 patchlevel 112) [x86_64-darwin14]
  - INSTALLATION DIRECTORY: /Users/danielkehoe/.rvm/gems/ruby-2.3.2@learn-rails
  - USER INSTALLATION DIRECTORY: /Users/danielkehoe/.gem/ruby/2.3.0
  - RUBY EXECUTABLE: /Users/danielkehoe/.rvm/rubies/ruby-2.3.2/bin/ruby
  - EXECUTABLE DIRECTORY: /Users/danielkehoe/.rvm/gems/ruby-2.3.2@learn-rails/bin
  - SPEC CACHE DIRECTORY: /Users/danielkehoe/.gem/specs
  - SYSTEM CONFIGURATION DIRECTORY: /Users/danielkehoe/.rvm/rubies/ruby-2.3.2/etc
  - RUBYGEMS PLATFORMS:
    - ruby
    - x86_64-darwin-14
  - GEM PATHS:
     - /Users/danielkehoe/.rvm/gems/ruby-2.3.2@learn-rails
     - /Users/danielkehoe/.rvm/gems/ruby-2.3.2@global
.
```

```
.
.
```

If you use RVM, gems are saved to a hidden **.rvm** folder in your user directory. A **global** subfolder contains the Bundler gem. If you've followed the instructions in the "Get Started" chapter to install Rails, the project-specific **learn-rails** subfolder contains the Rails gem. If you use Chruby or Rbenv instead of RVM, your gems will be stored in a different location.

Run the `gem which` command and you'll see where the gems live:

```
$ gem which bundler
/Users/danielkehoe/.rvm/gems/ruby-2.3.2@global/gems/bundler-1.12.5/lib/bundler.rb
$ gem which rails
/Users/danielkehoe/.rvm/gems/ruby-2.3.2@learn-rails/gems/railties-5.0.0.1/lib/rails.rb
```

These are details you'll never need to know, because Ruby on Rails handles it for you.

You'll never move or delete gems directly. Instead you'll manage gems using the Bundler utility. The key to Bundler is the Gemfile.

Gemfile

Every Rails application has a Gemfile. Earlier, I described Rails from the viewpoint of the "gem hunter," the developer who wants to assemble an application from the best open source components he or she can find. To the gem hunter, the Gemfile is the most important file in the application. It lists each gem that the developer wants to use.

The Gemfile provides the information needed by the Bundler utility to manage gems.

Bundler's `bundle install` command reads the Gemfile, then downloads and saves each listed gem to the hidden gem folder. Bundler checks to see if the

gem is already installed and only downloads gems that are needed. Bundler checks for the newest gem version and records the version number in the **Gemfile.lock** file. Bundler also downloads any gem dependencies and records the dependencies in the **Gemfile.lock** file. Between the Gemfile, with its list of gems that will be used by the application, and the **Gemfile.lock** file, with its list of dependencies and version numbers, you have a complete specification of every gem required to run the application. More importantly, when other developers install your application, Bundler will automatically install all the gems (including dependencies and correct versions) needed to run the application. When you deploy the application to production for others to use, automated deployment scripts (such as those used by Heroku) install all the required gems.

Bundler provides a **bundle update** command when we want to replace any gems with newer versions. If you run **bundle update**, any new gem versions will be downloaded and installed and the **Gemfile.lock** file will be updated. Be aware that updating gems can break your application, so only update gems when you have time to test and resolve any issues. You can run **bundle outdated** to see which gems are available in newer versions.

If you want to prevent your fellow developers (or yourself) from accidentally updating gems, you can specify a gem version number for any gem in the Gemfile. The Gemfile gives fine-grained control over rules for updating:

- **gem 'rails', '5.0.0'** is "absolute" only version 5.0.0 will be used

- **gem 'rails', '>= 5.0.0'** is "optimistic" any version newer than 5.0.0 will be used

- **gem 'rails', '~> 5.0.0'** is "pessimistic"

"Pessimistic" versioning needs some explanation. **~> 5.0.0** means use any version greater than 5.0.0 and less than 5.1 (any patch version can be used). **~> 5.0** means use any version greater than 5.0 and less than 6.0 (any minor version can be used).

In general, during development we only lock down any gem versions in the Gemfile if we know newer versions introduce problems.

Let's take a look at the Gemfile created by the `rails new` command.

Gemfile for a Rails Default Application

Open the **Gemfile** with your text editor:

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


# Bundle edge Rails instead: gem 'rails', github: 'rails/rails'
gem 'rails', '~> 5.0.0', '>= 5.0.0.1'
# Use sqlite3 as the database for Active Record
gem 'sqlite3'
# Use Puma as the app server
gem 'puma', '~> 3.0'
# Use SCSS for stylesheets
gem 'sass-rails', '~> 5.0'
# Use Uglifier as compressor for JavaScript assets
gem 'uglifier', '>= 1.3.0'
# Use CoffeeScript for .coffee assets and views
gem 'coffee-rails', '~> 4.2'
# See https://github.com/rails/execjs#readme for more supported runtimes
# gem 'therubyracer', platforms: :ruby

# Use jquery as the JavaScript library
gem 'jquery-rails'
# Turbolinks makes navigating your web application faster. Read more: https://github.com/turbol
gem 'turbolinks', '~> 5'
# Build JSON APIs with ease. Read more: https://github.com/rails/jbuilder
gem 'jbuilder', '~> 2.5'
# Use Redis adapter to run Action Cable in production
# gem 'redis', '~> 3.0'
# Use ActiveModel has_secure_password
# gem 'bcrypt', '~> 3.1.7'

# Use Capistrano for deployment
# gem 'capistrano-rails', group: :development

group :development, :test do
  # Call 'byebug' anywhere in the code to stop execution and get a debugger console
  gem 'byebug', platform: :mri
end
```

```
group :development do
  # Access an IRB console on exception pages or by using <%= console %> anywhere in the code.
  gem 'web-console'
  gem 'listen', '~> 3.0.5'
  # Spring speeds up development by keeping your application running in the background...
  gem 'spring'
  gem 'spring-watcher-listen', '~> 2.0.0'
end

# Windows does not include zoneinfo files, so bundle the tzinfo-data gem
gem 'tzinfo-data', platforms: [:mingw, :mswin, :x64_mingw, :jruby]
```

The file you see will be very similar. Some version numbers may be different if a newer Rails version was released since this was written.

The first line, **`source 'https://rubygems.org'`**, directs Bundler to use the rubygems.org server as a source for any gems.

Notice that the second uncommented line directs Bundler to use Rails and specifies a range of acceptable versions. In this case, the Gemfile indicates we can use any version between 5.0.0.1 and 6.0.

In the Gemfile you'll see the gems for a Rails default application, such as the sqlite3 database gem, which we described earlier. Other gems are commented out (the lines begin with the **`#`** character). These are suggestions and we can ignore them or remove them.

We won't use a database for our application but we'll keep the **`gem 'sqlite3'`** entry. Configuring Rails for no database is complicated; it is easier to keep the sqlite3 gem and not use it.

If you are developing your application on a computer using the Linux operating system, you may need to uncomment and use the statement **`gem 'therubyracer', platforms:  :ruby`**. Linux doesn't have a built-in JavaScript interpreter so you must install Node.js in your environment or else add the therubyracer gem to each project Gemfile. For help, see Install Ruby on Rails - Ubuntu.

It's wise to specify the Ruby version we're using. This is needed for automated deployment scripts such as those used by Heroku. We can add that to the Gemfile:

```
ruby '2.3.2'
```

If you add the Ruby version and remove the extra clutter in the **Gemfile** it will look like this:

```ruby
source 'https://rubygems.org'
ruby '2.3.2'
gem 'rails', '~> 5.0.0', '>= 5.0.0.1'

# Rails defaults
gem 'sqlite3'
gem 'puma', '~> 3.0'
gem 'sass-rails', '~> 5.0'
gem 'uglifier', '>= 1.3.0'
gem 'coffee-rails', '~> 4.2'
gem 'jquery-rails'
gem 'turbolinks', '~> 5'
gem 'jbuilder', '~> 2.5'
group :development, :test do
  gem 'byebug', platform: :mri
end
group :development do
  gem 'web-console'
  gem 'listen', '~> 3.0.5'
  gem 'spring'
  gem 'spring-watcher-listen', '~> 2.0.0'
end
```

Try it now. Replace the Gemfile with the simplified code above.

Adding Gems

I've identified several gems that will be useful for our tutorial application.

I learned about these gems from several places:

- Ruby Toolbox
- RubyFlow

- various blog posts

- example code and starter apps on GitHub

- recommendations from colleagues

We're adding these gems at the beginning of our development process since we already know which gems we'll need. On a real project, you'll often discover useful gems and add them to the Gemfile during the ongoing process of development.

Here are gems we'll add to the Gemfile:

- bootstrap-sass - front-end framework

- gibbon - access to the MailChimp API

- high_voltage - for static pages like "about"

- simple_form - forms made easy

We'll also add utilities that make development easier:

- better_errors - helps when things go wrong

- rails_layout - generates files for an application layout

Open your **Gemfile** and replace the contents with the following:

```
source 'https://rubygems.org'
ruby '2.3.2'
gem 'rails', '~> 5.0.0', '>= 5.0.0.1'

# Rails defaults
gem 'sqlite3'
gem 'puma', '~> 3.0'
gem 'sass-rails', '~> 5.0'
```

```ruby
gem 'uglifier', '>= 1.3.0'
gem 'coffee-rails', '~> 4.2'
gem 'jquery-rails'
gem 'turbolinks', '~> 5'
gem 'jbuilder', '~> 2.5'
group :development, :test do
  gem 'byebug', platform: :mri
end
group :development do
  gem 'web-console'
  gem 'listen', '~> 3.0.5'
  gem 'spring'
  gem 'spring-watcher-listen', '~> 2.0.0'
end

# learn-rails
gem 'bootstrap-sass'
gem 'gibbon'
gem 'high_voltage'
gem 'simple_form'
group :development do
  gem 'better_errors'
  gem 'rails_layout'
end
```

Notice that we've placed two gems inside a "group." Specifying a group for development or testing ensures a gem is not loaded in production, reducing the application's memory footprint. Rails let you specify groups for *development*, *test*, or *production*.

About the Rails Version

The version of Rails specified in your Gemfile should match the version that is installed in your gemset.

If you've got Rails 5.0, there's no need to make additional changes to the Gemfile. Any version beginning with 5.0, such as 5.0.1, will be fine.

If you have Rails 5.1 (which was not available when this was written), you must get a new version of this book. The newest available version of the book is listed on the README page of the learn-rails GitHub repository. If a newer

version of the book is not available, you can install Rails 5.0. See the articles Install Rails and Updating Rails for details about installing and switching between Rails versions.

Install the Gems

Each time you edit the Gemfile, you will run **bundle install** and restart your web server.

You've edited the Gemfile. Install the required gems on your computer:

```
$ bundle install
```

The **bundle install** command will download the gems from the rubygems.org server and save them to a hidden directory that is managed by the RVM gemset you've specified.

We'll see all the gems and their dependencies:

```
Fetching gem metadata from https://rubygems.org/
Fetching version metadata from https://rubygems.org/
Fetching dependency metadata from https://rubygems.org/
Resolving dependencies...
Using rake 11.3.0
Using concurrent-ruby 1.0.2
Using i18n 0.7.0
Using minitest 5.9.1
.
.
.
(many more gems not shown... you get the idea)
.
.
.
Bundle complete! 20 Gemfile dependencies, 73 gems now installed.
Use `bundle show [gemname]` to see where a bundled gem is installed.
```

You can use your text editor to view the contents of **Gemfile.lock** and you will see a detailed listing of every gem and each dependency, with version numbers.

There's no reason to edit a **Gemfile.lock** file; if it is ever in error, delete it and run **bundle install** to recreate it.

Run **gem list** to see all the gems that are loaded into the development environment:

```
$ gem list
```

The list of gems loaded in the environment is the same as the list specified in the **Gemfile.lock** file. Here's how it works. RVM makes a place for the gems to be stored (the RVM gemset); the **Gemfile** lists the gems you want to use; **bundle install** reads the Gemfile and installs the gems into the RVM gemset; the **Gemfile.lock** file records dependencies and version numbers; and **gem list** shows you the gems that are in the gemset and available for use.

Troubleshooting

If your development environment is set up correctly, there should be no difficulty installing gems with the **bundle install** command. If your development environment is not set up correctly, you may see error messages when Bundler attempts to install the Nokogiri gem. Nokogiri is often needed by other gems (it is a *dependency* of some gems) and Nokogiri can become a problem to install. Unlike most gems that are written in pure Ruby, parts of Nokogiri are written in the C language and must be compiled using system tools that vary with different operating systems. If you get an error while installing gems, and the message says, "An error occurred while installing nokogiri," ask for help on Stack Overflow.

Git

Let's commit our changes to the Git repository and push to GitHub:

```
$ git add -A
$ git commit -m "add gems"
$ git push origin master
```

After your first use of **git push origin master**, you can use the shortcut **git push**.

If you get a message:

```
fatal: Not a git repository (or any of the parent directories): .git
```

It indicates you are in a folder that has not been initialized with Git. You are probably not in your project directory. Use the Unix command **pwd** to see where you are.

If you get a message:

```
fatal: 'origin' does not appear to be a git repository
fatal: The remote end hung up unexpectedly
```

It shows that you can't connect to GitHub to push the changes. To investigate, enter:

```
$ git remote show origin
```

It is not absolutely necessary to use GitHub for this tutorial. We're only using it so you'll be familiar with the workflow of professional development.

We're ready to configure the application.

Chapter 10

Configure

Rails is known for its "convention over configuration" guiding principle. As applied, the principle reduces the need for many configuration files. It's not possible to eliminate all configuration files, however. Many applications require configuration of settings such as email account credentials or API keys for external services.

In our tutorial application, we'll need to store an API key to access MailChimp, which we'll use to add visitors' email addresses to a mailing list. We'll also need to store credentials so we can send email using the SendGrid transactional email service.

Rails provides the **config/secrets.yml** file for our configuration settings. Any variable that is set in the **config/secrets.yml** file can be used elsewhere in our Rails application, providing a single location for all our configuration variables.

Configuration Security

GitHub is a good place to store and share code. But when your repos are public, they are not a good place for secret account credentials. In fact, any shared Git repository, even a private repo, is a bad place to store email account credentials

or private API keys.

Operating systems (Linux, macOS, Windows) provide mechanisms to set local environment variables, as does Heroku and other deployment platforms. With a bit of Unix savvy, you can set environment variables using the Unix shell. Environment variables can be accessed from Rails applications and provide an ideal place to set configuration settings that must remain private.

For the best security, set credentials as Unix environment variables and only use Unix environment variables in the **config/secrets.yml** file.

The article Rails Environment Variables shows alternatives to using Unix environment variables, if for any reason you cannot set environment variables on your machine.

Videos

If you have subscribed, now's a good time to watch:

- UNIX Environment Variables

- Rails Environment Variables

About Environment Variables

Unix environment variables are typically set in a file that is read when starting an interactive shell. The *shell* is the program that gives us the command line interface we see in the Terminal or console application. Unix gives you a choice of shell programs (with names like *sh*, *bash*, *ksh*, and *zsh*); each has a slightly different way to set environment variables. The most common shell program is bash.

Let's find out what shell you are using:

```
$ echo $SHELL
/bin/bash
```

If you see **/bin/bash**, that's great! If not, you may have to do some research to find out how to set environment variables in your shell.

You might be surprised to see a dollar sign in the command. You don't type the first dollar sign (it is just the convention that indicates you are entering a Unix command). You'll type **echo $SHELL** to ask the operating system to show the variable **SHELL**. The dollar sign in the command tells Unix to return a variable named **SHELL**. Try typing **echo SHELL** without the dollar sign and you'll see **echo** just displays what you type.

When you open a console window, the bash shell reads a configuration file in your user home directory. You can use a Unix command to list all the files in your user home directory (the **~** "tilde" character represents your home directory):

```
$ ls -1pa ~
.
.
.
.bash_profile
.
.
.
```

On the Mac, you'll see **.bash_profile**. On Linux systems, you'll see **.bashrc**.

Viewing Hidden Files

The files **.bash_profile** or **.bashrc** are hidden in the file browser. You can force the Mac to display hidden files by entering the following command in the Terminal application:

```
defaults write com.apple.finder AppleShowAllFiles TRUE; killall Finder
```

Hidden files will appear in gray in the Finder window.

Use your text editor (Atom or Sublime) to open the **.bashrc** or **.bash_profile** file.

Open either file and you'll likely find a command such as:

```
export PATH=~/.bin:$PATH
```

That is a command that sets the **PATH** environment variable. The command might not be exactly the same but it is likely you will see some **export** commands.

You can add the environment variables anywhere in the file. For convenience, add the environment variables near the end of the file, above any existing **EXPORT** statement.

You should use quotes to surround configuration values (credentials) in the **.bash_profile** or **.bashrc** files.

If you don't have a **.bash_profile** or **.bashrc** file in your user home directory, you can create one.

Set Environment Variables

You'll set the following environment variables in your **.bashrc** or **.bash_profile** file:

- SENDGRID_USERNAME

- SENDGRID_PASSWORD

- MAILCHIMP_API_KEY

- MAILCHIMP_LIST_ID

- OWNER_EMAIL

Here are details.

SendGrid

You'll need your SendGrid username and password. The credentials are the same you use to sign in to the SendGrid website.

Add your SendGrid username and password to your **.bash_profile** or **.bashrc** file:

```
export SENDGRID_USERNAME="example"
export SENDGRID_PASSWORD="secret"
```

Obviously, change "example" and "secret" to your own credentials.

MailChimp

When visitors sign up to receive a newsletter, we'll add them to a MailChimp list. Add an environment variable for the MailChimp API key: **MAILCHIMP_API_KEY**. Sign in to MailChimp to get your API key. Click your name at the top of the navigation menu, then click "Account." Click "Extras," then "API keys." You have to generate an API key; MailChimp doesn't create one automatically. The MailChimp API key is a long string of characters like a secret code that works like a password. Enter it in your **.bash_profile** or **.bashrc** file:

```
export MAILCHIMP_API_KEY="Your_MailChimp_API_Key"
```

You'll need to create a MailChimp mailing list in preparation for our "Mailing List" chapter. Have you already created a MailChimp mailing list? If not, the MailChimp "Lists" page has a button for "Create List." The list name and other details are up to you.

We'll need the **MAILCHIMP_LIST_ID** for the mailing list you've created. To find the list ID, on the MailChimp "Lists" page, click the "down arrow" for a menu and click "Settings." At the bottom of the "List Settings" page, you'll find the unique ID for the mailing list.

```
export MAILCHIMP_LIST_ID="Your_List_ID"
```

Your environment variables are set up to use MailChimp.

Owner Email

You'll send email messages to this address when a visitor submits a contact request form. Set **OWNER_EMAIL** with an email address where you receive mail.

```
export OWNER_EMAIL="example@example.com"
```

Enter an email address and your environment variables will be set up with the site owner email address.

Restart the Terminal Session

Close and reopen your terminal to make sure the environment is updated with any recent changes.

Troubleshooting

Check that the SendGrid user name is set in your Unix environment variables:

```
$ echo "$SENDGRID_USERNAME"
```

You should see your SendGrid user name in the console response. Make sure you've used underscores consistently and you've used **SENDGRID_USERNAME** not **SENDGRID_USER_NAME**.

If you have trouble, remember to close and reopen your terminal to make sure the environment includes any recent changes.

On Linux, if you've entered the environment variables in your **.bashrc** file but they don't seem to work, try setting them in your **.bash_profile** file instead.

If you've set up Unix environment variables but **echo "$SENDGRID_USERNAME"** doesn't return the correct variable in the console, you may have a problem with the way you've set Unix environment variables. Most computers use the bash shell and you can set environment variables in your **.bash_profile** or **.bashrc** files. But not every system is alike. If it seems Unix environment variables are not working, you may have to find a colleague who can help you troubleshoot. If you are having problems, you can continue with the tutorial and add the credentials directly to the **config/secrets.yml** file.

The Secrets File

Use your text editor to add the Unix environment variables to the file **config/secrets.yml**:

```
# Be sure to restart your server when you modify this file.

# Your secret key is used for verifying the integrity of signed cookies.
# If you change this key, all old signed cookies will become invalid!
```

```
# Make sure the secret is at least 30 characters and all random,
# no regular words or you'll be exposed to dictionary attacks.
# You can use `rails secret` to generate a secure secret key.

# Make sure the secrets in this file are kept private
# if you're sharing your code publicly.

development:
  email_provider_username: <%= ENV["SENDGRID_USERNAME"] %>
  email_provider_password: <%= ENV["SENDGRID_PASSWORD"] %>
  domain_name: example.com
  mailchimp_api_key: <%= ENV["MAILCHIMP_API_KEY"] %>
  mailchimp_list_id: <%= ENV["MAILCHIMP_LIST_ID"] %>
  owner_email: <%= ENV["OWNER_EMAIL"] %>
  secret_key_base: very_long_random_string

test:
  secret_key_base: very_long_random_string

# Do not keep production secrets in the repository,
# instead read values from the environment.
production:
  email_provider_username: <%= ENV["SENDGRID_USERNAME"] %>
  email_provider_password: <%= ENV["SENDGRID_PASSWORD"] %>
  domain_name: example.com
  mailchimp_api_key: <%= ENV["MAILCHIMP_API_KEY"] %>
  mailchimp_list_id: <%= ENV["MAILCHIMP_LIST_ID"] %>
  owner_email: <%= ENV["OWNER_EMAIL"] %>
  secret_key_base: <%= ENV["SECRET_KEY_BASE"] %>
```

Be sure to use spaces, not tabs. Make sure there is a space after each colon and before the value for each entry or you will get a message "Internal Server Error: mapping values are not allowed" when you start the web server.

You used quotes to surround configuration values in the **.bashrc** or **.bash_profile** files. Here, in the **config/secrets.yml** file, you don't need quotes when you are importing Unix environment variables.

Domain Name

We'll need a domain name when we configure email for delivery in production. For development, use `example.com`. If you have your own domain name, you

can use that instead. There's no need to keep the **`domain_name`** configuration variable secret, so we don't need to set it in a Unix environment variable.

You can decide for yourself if the **`owner_email`** variable really needs to be secret. Just for caution, I'm suggesting you set it as a Unix environment variable.

Securing the Secrets File

Some developers take steps to prevent the **config/secrets.yml** file from being checked into Git. To prevent the file from being saved to your repo you could add the filename to the **.gitignore** file in your application root directory.

However, you don't need to keep the **config/secrets.yml** file from being checked into Git if you've used Unix environment variables in the **config/secrets.yml** file. If you only reveal the **`SECRET_KEY_BASE`** used for development or testing, and no one can access your development machine, no useful secrets will be revealed in your GitHub repo.

When you deploy to Heroku, the **config/secrets.yml** file must be in your Git repository. For that reason, I suggest you save the file in your Git repo and keep your secrets safe by using environment variables.

Troubleshooting

Remember, in YAML files (with the file extension **.yml**), indentation is required (your application will break without it).

Be sure to use spaces, not tabs. Make sure there is a space after each colon and before the value for each entry.

If you have trouble setting Unix environment variables, you can add credentials directly to the **config/secrets.yml** file. If you do so, you should not check the file into Git until you've deleted the secrets from the file.

Replace the following if you are troubleshooting:

```
development:
  email_provider_username: <%= ENV["SENDGRID_USERNAME"] %>
  email_provider_password: <%= ENV["SENDGRID_PASSWORD"] %>
```

with:

```
development:
  email_provider_username: example
  email_provider_password: 's#cr*t'
```

In a YAML file, you do not need quotes unless your string contains special characters. If your password contains any of these characters you should surround the string with single quotes:

```
: { } [ ] & * # ? | - < > = ! % @ \
```

Again, DON'T CHECK THE FILE INTO GIT if you've hardcoded your credentials directly in the **config/secrets.yml** file.

The article Rails Environment Variables shows alternatives to using Unix environment variables, if for any reason you cannot set environment variables on your machine.

Secret Key Base

It's not necessary to set **SECRET_KEY_BASE** as an environment variable on the computer you use for development. Rails generates a unique **SECRET_KEY_BASE** in the **config/secrets.yml** file each time you create a new Rails application and you don't need to replace it. If someone sees the **SECRET_KEY_BASE** in the

config/secrets.yml file in your GitHub repo, there isn't anything they can do with it, since they don't have access to your local machine.

For your future reference, in case you want to change the **SECRET_KEY_BASE**, here's how. Go to your Rails application directory and create a new secret token:

```
$ rails secret
very_long_random_string
```

And, if you wish, add it to your **.bash_profile** or **.bashrc** file:

```
export SECRET_KEY_BASE="very_long_random_string"
```

You should always use the environment variable **<%= ENV["SECRET_KEY_BASE"] %>** in the production section of your **config/secrets.yml** file, otherwise, someone who sees the secret token in your GitHub repo can gain access to your application in production. You'll set the environment variables for production when you deploy to Heroku.

Configure Email

Email messages are visible in the console and the log file when you test the application. If you don't want to actually send email, you can skip this step. But it's more fun when your application can actually send email.

You can learn more in the article Send Email with Rails.

Connect to an Email Server

Web servers don't send email. Our Rails application has to connect to an email server (also known as a mail transfer agent or "mail relay"). In the early days of

the Internet, an experienced system administrator could set up an SMTP server
to distribute email. Now, because of efforts to reduce spam, it is necessary
to use an established email service to ensure deliverability. In production, for
high volume transactional email and improved deliverability, it is best to use a
service such as SendGrid. Alternatives are:

- Amazon SES (Simple Email Service)

- Mailgun

- Mailjet

- Mandrill

- PostageApp

- Postmark

- SparkPost

For our tutorial application, we'll connect to SendGrid to send email.

For convenience during development, some developers use their own Gmail
account to send email. Google has increased security measures for Gmail, so
it is difficult to use Gmail to send email from a Rails application. SendGrid is
easier to set up and you're more likely to use it for a real application. That's
why we'll use it.

In the file **config/environments/development.rb**, near the end of the file, find
the statement:

```
config.assets.debug = true
```

Immediately following, add this:

```
config.action_mailer.smtp_settings = {
  address: "smtp.sendgrid.net",
  port: 587,
  domain: Rails.application.secrets.domain_name,
  authentication: "plain",
  enable_starttls_auto: true,
  user_name: Rails.application.secrets.email_provider_username,
  password: Rails.application.secrets.email_provider_password
}
# ActionMailer Config
config.action_mailer.default_url_options = { :host => 'localhost:3000' }
config.action_mailer.delivery_method = :smtp
config.action_mailer.raise_delivery_errors = true
# Send email in development mode?
config.action_mailer.perform_deliveries = true
```

You can use port 25, 587, or 2525 (some ISPs restrict connections on port 25).

It's important to add these changes in the body of the configuration file, before the **end** keyword. The order isn't important but don't add the configuration statements after the **end** keyword.

Notice that we are using configuration variables that are set in the **config/secrets.yml** file:

- Rails.application.secrets.email_provider_username

- Rails.application.secrets.email_provider_password

We could "hard code" a username and password here but that would expose confidential data if your GitHub repository is public. Using configuration variables that are set in the **config/secrets.yml** file keeps your secrets safe.

Again, if you need to troubleshoot, you can enter the SendGrid username and password directly in this file instead of the configuration variables. But for security, don't commit to Git with the password hardcoded in this file.

Perform Deliveries in Development

If you want to send real messages when you test the application in development mode, modify the file **config/environments/development.rb**.

After the code you just added, add the statement:

```
# Send email in development mode?
config.action_mailer.perform_deliveries = true
```

This changes the configuration to send email when you're working on the application.

Make sure any code you've added to the **config/environments/development.rb** file is placed before the final **end** keyword. If you add code after the final **end** keyword, your application will fail with errors when you start the web server.

Later, after we add a contact form to the tutorial application, the application will be ready to send email messages.

Git

Make sure you're in your application root directory.

Let's commit our changes to the Git repository and push to GitHub:

```
$ git add -A
$ git commit -m "add configuration"
$ git push
```

We're ready to create a home page for the application.

Chapter 11

Static Pages and Routing

A Rails application can deliver static web pages just like an ordinary web server. The pages are delivered fast and no Ruby code is required. We'll look at simple static pages and learn about Rails routing before we explore the complexities of dynamic web pages in Rails.

Add a Home Page

Make sure you are in your project directory.

Start the application server:

```
$ rails server
```

Open a web browser window and navigate to http://localhost:3000/. You'll see the Rails default information page.

Use your text editor to create and save a file **public/index.html**:

```
<h1>Hello World</h1>
```

Refresh the browser window and you'll see the "Hello World" message.

The Rails application server looks for any pages in the **public** folder by default.

If no filename is specified in the URL, the server will attempt to respond with a file named **index.html**. This is a convention that dates to 1993; if no filename was specified, one of the first web servers ever built (the NCSA httpd server) would return a list of all files in the directory, unless a file named **index.html** was present. Since then, **index.html** has been the default filename for a home page.

Routing Error

What happens when no file matches the requested web address?

Enter the URL http://localhost:3000/about.html in your browser.

You'll see an error page that shows a routing error.

If you are using Cloud9, add "/about.html" to the URL in the preview browser window.

Add an About Page

Use your text editor to create and save a file **public/about.html**:

```
<h1>About</h1>
```

Visit the URL http://localhost:3000/about.html in your browser. You'll see the new "About" page.

By the way, you've just done test-driven development (TDD).

Introducing TDD

With test-driven development, a developer tests behavior before implementing a feature, expecting to see an error condition. Then the developer implements the feature and sees a successful result to the test. That's exactly what you've done, in the simplest way.

Beginners tend to think TDD is scary and complicated. Now that you've experienced a simple form of TDD, maybe it won't be intimidating. Real TDD means writing tests in Ruby before implementing features, but the principle is the same.

Introducing Routes

The guiding principle of "convention over configuration" governs Rails routing. If the web browser requests a page named "index.html", Rails will deliver the page from the **public** folder by default. No configuration is required. But what if you want to override the default behavior? Rails provides a configuration file to control web request routing.

Remove the **public/index.html** file:

```
$ rm public/index.html
```

If you've got only one terminal window open, you'll have to stop the Rails server with Control-c to get your terminal prompt. Here is where it is helpful to have two terminal sessions going in different tabs.

Now let's set the "About" page as the home page.

Open the file **config/routes.rb**. Remove all the comments and replace the file with this:

```ruby
Rails.application.routes.draw do
  root to: redirect('/about.html')
end
```

This snippet of Rails routing code takes any request to the application root (http://localhost:3000/) and redirects it to the **about.html** file (which is expected to be found in the **public** folder).

There is no need to restart your application server to see the new behavior. If you need to start the server:

```
$ rails server
```

Visit the page http://localhost:3000/. You'll see the "About" page.

If you still see the "Hello World" page, you didn't remove the the the **public/index.html** file. Rails will use the index file in the public folder before it checks the routing file for a redirect.

You've just seen an example of Rails magic. Some developers complain that the "convention over configuration" principle is black magic. It's not obvious why pages are delivered from the **public** folder; it just happens. If you don't know the convention, you could be left scratching your head and looking for the code that maps http://localhost:3000/ to the **public/index.html** file. The code is buried deep in the Rails framework. However, if you know the convention and the technique for overriding it, you have both convenience and power at your disposal.

Using the "About" Page

We've created an "About" page so we can learn about routing.

For the next chapter, we'll use the static "About" page to investigate how a web application works.

Later in the tutorial we'll create a new "About" page using a different approach.

Chapter 12

Request and Response

You've configured the tutorial application, created static pages, and seen the magic of Rails routing.

In this chapter, we'll investigate the web request-response cycle and look at the model-view-controller design pattern so you'll be prepared to build a dynamic home page.

Investigating the Request Response Cycle

Remember, at its core, the World Wide Web is nothing more than web browsers that request files from web servers.

Web browsers make *requests*. A web server *responds* to a request by sending an HTML file. Depending on the headers in the HTML file, the web browser may make additional requests and get additional CSS, JavaScript, and image files.

The beauty and simplicity of the World Wide Web architecture, as conceived by Tim Berners-Lee in 1990, is that the web is nothing more than a request from a web browser and a response from a web server. Some web pages now include streaming video, or music, requiring an open "pipe" between the web server

and the web browser, but even so, an initial request-response cycle delivers the page that sets up the stream.

We can reduce the mystery of how the web works to its simplest components when we investigate the request-response cycle. We'll see that everything that happens in a web application takes place within the flow of the request-response cycle.

Let's look at the request-response cycle.
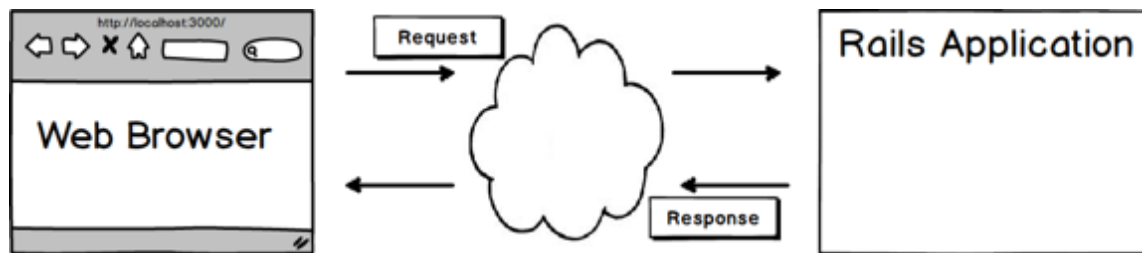
Figure 12.1: The request-response cycle.

Inside the Browser

We can see the actual request, and the actual response, by using the diagnostic tools built into the web browser.

Start the application server if it is not already running:

```
$ rails server
```

Developers use various web browsers during development. I'll provide instructions for Chrome, since it is the most popular. Even if you prefer Mozilla Firefox or Apple Safari, try this in Chrome, so you can follow along with the text.

Start our investigation by putting Chrome into "Incognito Mode" with Command-Shift-N (on a Mac). On Linux, use Ctrl-Shift-N to get in incognito mode with

Chrome. Alternatively, you can clear the browser cache. This clears any files that were previously cached by the browser.

The Developer Tools View is your primary diagnostic tool for front-end (browser-based) development, including CSS and JavaScript.

In Chrome on macOS, press Command-Option-I to open the *Developer Tools View* in a section of the browser window. Alternatively, you can find the menu item under View/Developer/Developer Tools.

In Chrome on Windows or Linux platforms, press Shift-Ctrl-I or select Menu/Tools/Developer Tools.

Select the Network tab in the Developer Tools View.

Initiate the request-response cycle by visiting the "About" page at http://localhost:3000/about.html

In the Developer Tools View, you'll see files received by the browser from the web server. There is only one: "about.html". This is the file that the browser evaluates to display a web page.

Click the "about.html" file icon. Then click the tab "Headers." The diagnostic window shows the entire request sent to the server and the entire response received by the browser.

The request is composed of:

- request URL (http://localhost:3000/about.html)

- request method (GET)

- request headers (including cookies and User Agent identifier)

The response is composed of

- status code (200 OK or 304 Not Modified)

- response headers (including date/time and server identifier)
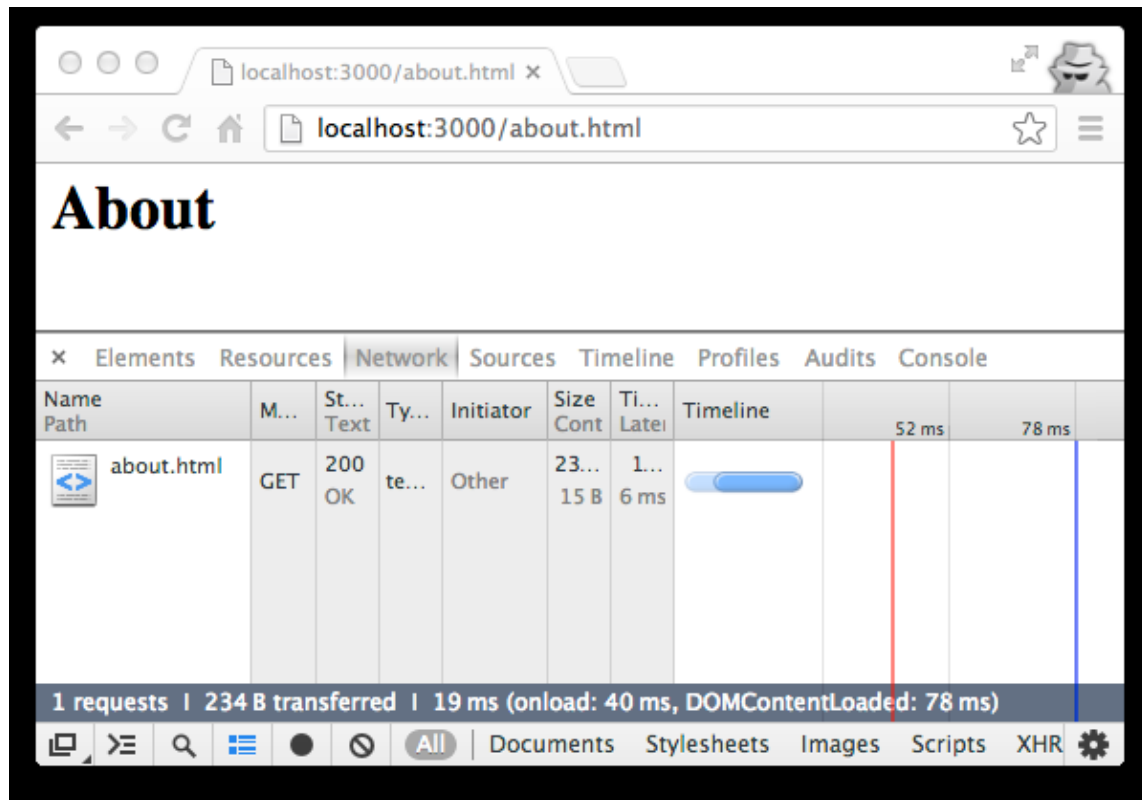
Figure 12.2: Viewing a request in the Developer Tools View.

- HTML

You can see the HTML sent to the browser by clicking the Preview or Response tabs in the diagnostic view.

Now try requesting the home page by entering the URL http://localhost:3000/.

You'll see the server returns two files. The first, "localhost", contains a redirect code "301 Moved Permanently" that tells the browser to request the "about.html" file. The second file is the "about.html" file. You may see the status code "200 OK" the first time the file is requested. On subsequent requests, you'll see the "304 Not Modified" code, indicating that the file hasn't changed and the browser should use the file that has been previously cached.

Here's the point of the exercise: The browser's diagnostic view shows all the data exchanged between the browser and server. You're looking at everything that passes through the plumbing.

Inside the Server

The browser's diagnostic view doesn't show you what happens on the server. For that, go to the server logs or the console window.

```
Started GET "/" for ::1 at ...
```

Notice how the diagnostic messages in the console window match the headers in the browser diagnostic view. The browser's "Request Method:GET" matches the server's "Started GET."

Notice there are no console log messages for pages delivered from the **public** folder.

Soon we'll see much more in the console window, after we've built a dynamic web page that is assembled by the application server.

How the Browser Works

What happens after the browser receives a response from the server?

The response is not complete until all files are received (or the browser reaches a time-out limit). Modern browsers retrieve files asynchronously; the order and location of the files in the initial HTML file doesn't matter because the browser will try to load all the files before displaying the page.

Document Object Model

When the web browser receives an HTML file, it creates an internal representation of the page in computer memory, called the Document Object Model (DOM). It provides a structural representation of the document. The DOM works like an API for HTML documents, allowing you to modify the content and visual presentation of the page by using JavaScript.

Later in the tutorial, we'll see how a JavaScript library such as jQuery can be used to do things like hiding or revealing HTML elements on a page by manipulating the DOM.

Rendering

Each time the DOM changes, the browser engine converts the DOM to a visual representation of the page and renders it in the browser window.

Knowing about the DOM will help you understand what happens in the browser when it receives a web response. And understanding the DOM will help you work with JavaScript for front-end programming. But in this book, our focus is on building a server-side web application in Rails. Let's see how a Rails application responds to a web request.

How the Application Works

Now that we've investigated the request-response cycle, let's dig deeper to understand what happens inside the Rails application in response to a browser request. To do so, we'll need to understand the model–view–controller concept.

Video Option

This eight minute video introduces the model–view–controller concept:

- Model View Controller in Rails

The Model View Controller Concept

The model–view–controller concept is key to understanding how a Rails application responds to a browser request.

Here is a diagram that shows what happens in the server during the request-response cycle.

You learned earlier that, from the perspective of a software architect, Rails is organized to conform to the model–view–controller software design pattern. This enforces "separation of concerns" to keep code manageable and organized. The MVC design pattern is optimal for web applications and is a central organizing principle for Rails.

The MVC design pattern originated in the design of desktop applications. "Model" classes manipulated data; "view" classes created the user interface; and a "controller" class responded to user interaction.

Some computer scientists feel the architecture of web applications doesn't quite match the original MVC design pattern of desktop applications. We can see the reason for the quibble in the next diagram. The diagram shows the MVC architecture as part of the Rails software stack.

At the base of the stack is the web browser. A request flows upward through the layers and encounters the router which dispatches the request to an appropriate controller.

In a Rails application, there is a single routing file, **config/routes.rb**, and multiple controllers, models, and views.

Considering the importance of the router, perhaps we should call our Rails

architecture the *RCMV, or Routing-Controller-Model-View, pattern*. Despite
the quibble about nomenclature, the architecture is well understood and used
by all Rails developers.

Here's the step-by-step walk-through of what happens.

When the web browser makes a request, a *router* component will check the
config/routes.rb file and determine which *controller* should handle the request,
based on the web address and HTTP protocol. The controller will obtain any
needed data from a *model*. After obtaining data, the controller will render a
response combining data from the model with a *view* component that provides
markup and layout. The response is an HTML file that the controller assembles
for the browser to display.

The model, view, and controller are files you create containing Ruby code.
Each file has a certain structure and syntax based on foundation model, view,
and controller classes defined in the Rails framework. The model, view, and
controller classes you create will *inherit* behavior from parent classes that are
part of the framework, so you will have less code to write yourself.

In most Rails applications, a **model** obtains data from a database, though some
models obtain data from a remote connection to another server. For example, a
User model might retrieve a user name and email address from a local database.
A User model could also obtain a user's recent tweets from Twitter or a user's
hometown from Facebook. The controller can obtain data from more than one
model if necessary.

A **controller** can have more than one *action*. For example, a User controller
might have actions to display a list of users, or add or delete a user from a
list. The **config/routes.rb** file matches a web request to a controller action. In
the software architects' terminology, each action is a *method* of the controller
class. We use the terms *action* and *method* interchangeably when we talk about
a Rails controller; to be precise, controller actions are implemented as methods.

In practice, Rails developers try to limit controllers to seven standard actions:
`index`, `show`, `new`, `create`, `edit`, `update` and `destroy` actions. A con-
troller that offers these actions is said to be "RESTful" (a term that refers to

representational state transfer, another software design abstraction). It's not important to understand the abstract principles of RESTful design; recognizing the term and knowing that Rails controllers have seven standard actions is sufficient for beginners.

A **view** file combines Ruby code with HTML markup. Typically there will be a view file associated with each controller action that displays a page. An index view might show a list of users. A "show" view might provide details of a user's profile. View files look much like ordinary HTML files but typically contain data in the form of Ruby variables. Often you'll see Ruby statements such as blocks that iterate through lists to create tables. Following the "separation of concerns" principle, it is considered good practice to limit Ruby code in view files to only displaying data; anything else belongs in a model.

Not every controller action has its own view file. In many controllers, on completion, the **destroy** action will redirect to the index view, and **create** will redirect to either **show** or **new**.

This conceptual overview will be easier to grasp when you actually see the code for a model, view, and controller. We'll create model, view, and controller files in the next chapter.

Remove the About Page

We've been using the static "About" page to investigate the request-response cycle.

We're done, so delete the file **public/about.html**:

```
$ rm public/about.html
```

Make sure you've removed the **public/index.html** file as suggested earlier:

```
$ rm public/index.html
```

Earlier, we set up the **config/routes.rb** file. You can leave it in place. We'll change it in the next chapter.

Now we'll look at ways to implement the home page using the full power of Rails.
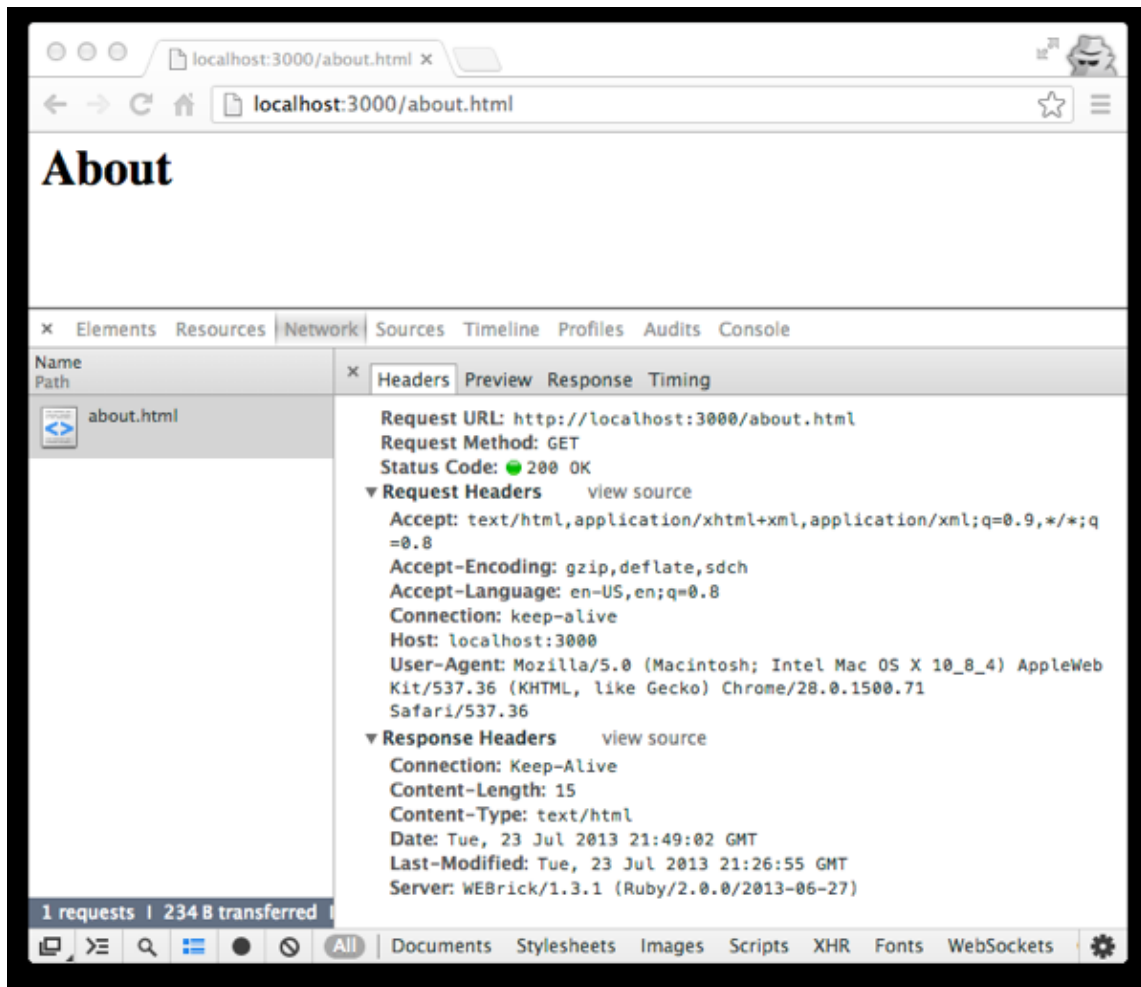
Figure 12.3: Viewing request headers in the Developer Tools View.
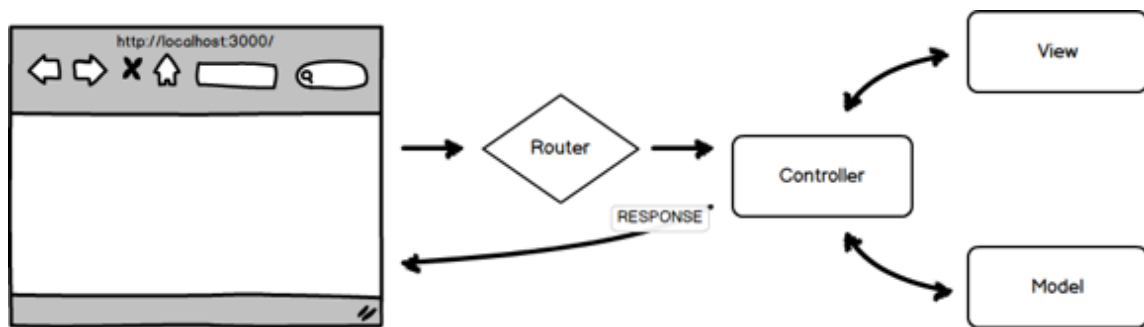
Figure 12.4: Model–View–Controller in Rails.
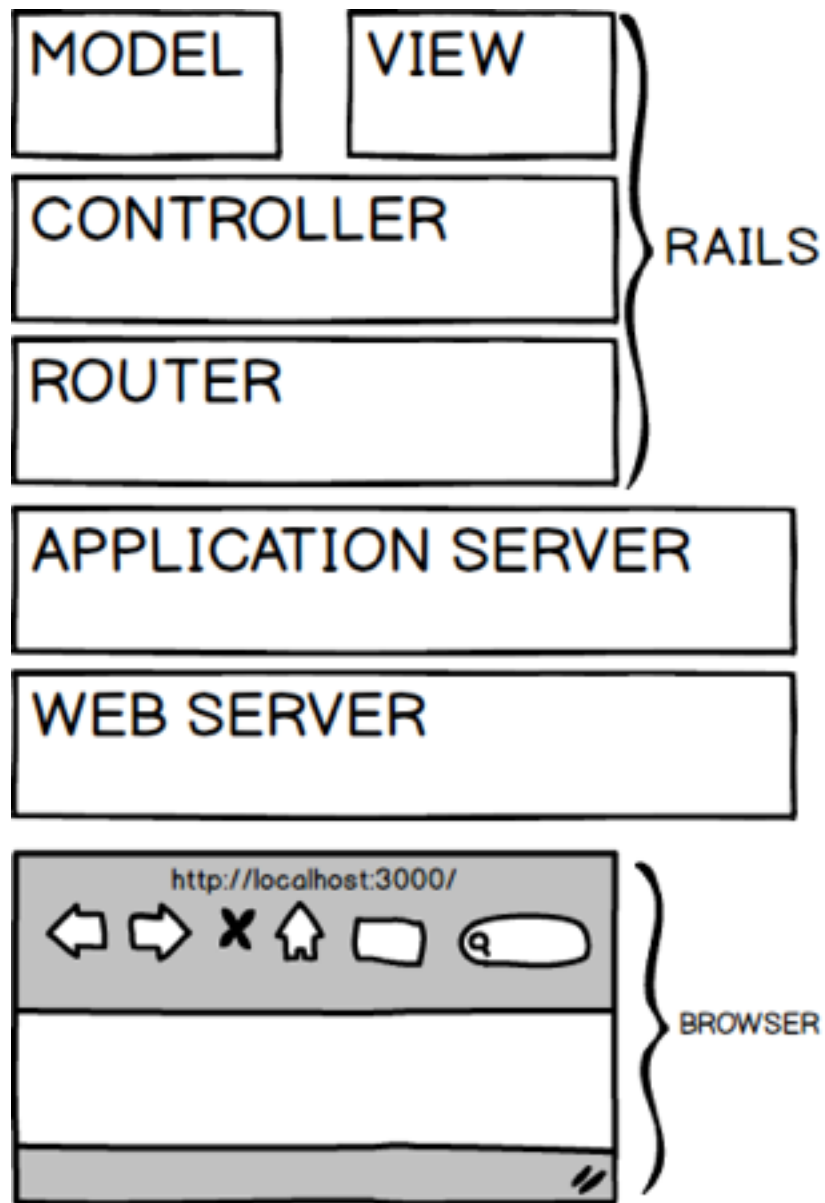
Figure 12.5: Model–View–Controller stack in Rails.

Chapter 13

Dynamic Home Page

Earlier, we saw how Rails can deliver simple static web pages.

Here we'll build a dynamic home page, illustrating basic concepts you'll need to understand Rails.

User Story

We'll plan our work with a user story:

```
*Birthday Countdown*
As a visitor to the website
I want to see the owner's name
I want to see the owner's birthdate
I want to see how many days until the owner's next birthday
In order to send birthday greetings
```

This silly home page will help us explore Rails and learn about the Ruby language.

Our goal is to build a practical web application that you can really use. Later we'll replace this silly home page with a useful web page that encourages visitors to sign up for a mailing list.

Routes, Model, View, and Controller

We'll use the model-view-controller design pattern as we build our new home page.

First, we'll set up a route so a request URL gets directed to the appropriate controller.

We'll set up a model so we obtain data we need for the home page.

We'll set up a view that contains the HTML needed to display our home page.

And finally, we'll create a controller that responds to the request, obtaining data from the model and rendering the view, sending a response to the web browser.

We can create the routes, model, view, and controller in any order. All must exist before our web application will respond to a request for a home page. In this tutorial, I've chosen to create the routes, model, view, and controller in an order that is convenient for learning.

The Name Game

Much of the art of programming lies in choosing suitable names for our creations.

We'll need a model as a source for data about the site owner. Choosing the most obvious name, we'll call it the Owner model:

- Owner - the file will be **app/models/owner.rb**

What about a name for the controller that will render our home page? How about "Home controller" or "Welcome controller?" Those names are acceptable. But if we consider our user story, the name "Visitors controller" is best. A visitor is the actor, so "Visitors controller" is appropriate:

- VisitorsController - the file will be **app/controllers/visitors_controller.rb**

Later we'll see this is a good choice because we'll create a Visitor model to handle data about the website visitor. In Rails, there is often a model with the same name as a controller (though a controller can use data from multiple models).

Naming Conventions

Rails is picky about class names and filenames. That's because of the "convention over configuration" principle. By requiring certain naming patterns, Rails avoids complex configuration files.

Before we look at class and filename conventions, here's a note about typographic terminology:

- a string is a sequence of characters

- you're looking at an example of lowercase strings separated by spaces (words!)

- Titlecase means there is an Initial Capital Letter in a string

- CamelCase contains a capital letter in the middle of a string

- snake_case combines words with an underscore character instead of a space

When you write code, you'll follow rules for class names:

- **`class Visitor`** - the model class name is capitalized and singular

- **`class VisitorsController < ApplicationController`** - for a controller, combine a pluralized model name with "Controller" in Camel-Case

Here are the rules for filenames. They are always lowercase, with words separated by underscores (snake_case):

- the model filename matches the model class name, but lowercase, for example **app/models/visitor.rb**

- the controller filename matches the controller class name, but snake_case, for example **app/controllers/visitors_controller.rb**

- the views folder matches the model class name, but plural and lowercase, for example **app/views/visitors**

At first the rules may seem arbitrary, but with experience they will make sense. The rule about no capital letters or spaces in filenames has its origins in computer antiquity.

If you stray from these naming conventions, you'll encounter unexpected problems and frustration.

Routing

We'll create the route before we implement the model and controller.

Open the file **config/routes.rb**. Replace the contents with this:

```ruby
Rails.application.routes.draw do
  root to: 'visitors#new'
end
```

Any request to the application root (http://localhost:3000/) will be directed to the VisitorsController `new` action.

Don't be overly concerned about understanding the exact syntax of the code. It will be become familiar soon and you can look up the details in the reference documentation, RailsGuides: Routing from the Outside In.

In general, when you change a configuration file you must restart your application server. However, the **config/routes.rb** file is an exception. You don't need to restart the server after changing routes.

If you need to start the server:

```
$ rails server
```

Visit the page http://localhost:3000/. You'll see an error message because we haven't implemented the controller. The error message, "uninitialized constant VisitorsController," means Rails is looking for a VisitorsController and can't find it.

Model

Most Rails models obtain data from a database. When you use a database, you can use the `rails generate model` command to create a model that inherits from the ActiveRecord class and knows how to connect to a database.

To keep things simple, our tutorial application doesn't need a database. Instead of inheriting from ActiveRecord, we create a Ruby class with methods that return the owner's name, birthdate, and days remaining until his birthday. This simple class provides an easy introduction to Ruby code.

Create a file **app/models/owner.rb**:

```ruby
class Owner

  def name
    name = 'Foobar Kadigan'
  end

  def birthdate
    birthdate = Date.new(1990, 12, 22)
  end

  def countdown
    today = Date.today
    birthday = Date.new(today.year, birthdate.month, birthdate.day)
    if birthday > today
      countdown = (birthday - today).to_i
    else
      countdown = (birthday.next_year - today).to_i
    end
  end

end
```

This is your first close look at Ruby code. The oddest thing you'll see is the owner's name, "Foobar Kadigan." Everything else will make sense with a bit of explanation.

Keep in mind that we are using a text file to create an abstraction that we can manipulate in the computer's memory. Software architects call these abstractions *objects*. In Ruby, everything we create and manipulate is an *object*. To distinguish one object from another, we define it as a *class*, give it a *class name*, and add behavior in the form of *methods*.

The first line **class Owner** defines the class and assigns a name. At the very end of the file, the **end** keyword completes the class definition.

We define three methods, starting with **def** (for "method definition") and ending with **end**.

- **def name** ... **end**

- **def birthdate** ... **end**

- **def countdown** ... **end**

Each method contains simple Ruby code that assigns data to a variable. Later, we'll retrieve the data for use in our view file by *instantiating* the class and *calling* a method. Don't be discouraged by the software architects' terminology; the concepts are simple and we'll soon see everything in action.

Ruby makes it easy for a method to *return* data when called; the value assigned by the last statement will be delivered when the method is called.

Looking more closely at the Ruby code inside the method definitions, you'll see Ruby uses the **=** (equals) sign to assign values to a variable. The variable is named on the left side of the equals sign; a value is assigned on the right side. We call the equals sign an *assignment operator*.

We can assign any value to a variable, including a *string* (a series of characters that can be a word or name) such as "Foobar Kadigan." Ruby recognizes a string when characters are enclosed in single or double quotes. Not surprisingly, a number also can be assigned to a variable, either a whole number (an *integer*) or a decimal fraction (a *float*).

More interestingly, any Ruby object can be assigned to a variable. That helps us "move around" any object very easily, giving us access to the object's class methods anywhere we use the variable. We can create our own objects, as we have by creating the Owner class. Or we can use the library of objects that are supplied with Ruby. Ruby's prefabricated objects are defined by the Ruby API (*application programming interface*); essentially the API is a catalog of prebuilt classes that are building blocks for any application. The Rails API gives us additional classes that are useful for web applications. Learning the syntax of Ruby code gets you started with Ruby programming; knowing the API classes leads to mastery of Ruby.

The **Date** class is provided by the Ruby API. It is described in the Ruby API reference documentation. The **Date** class has a **Date.new** method which *instantiates* (creates) a new date when supplied with year, month, and day *parameters*. You can see this syntax when we assign **Date.new(1990, 12, 22)** to the **birthdate** variable.

Note that Ruby has specific expectations about the syntax of numbers. The `Date.new(...)` method expects integers. Imagine a September birthday. You must use `Date.new(1990, 9, 22)`. If you enter a date in the format `Date.new(1990, 09, 22)`, you'll get a syntax error "Invalid octal digit" when you test the application. Ruby expects numbers that begin with zero to be octal numbers; you'll get an error because octal numbers can't contain the digit "9."

Our `countdown` method contains the most complex code in the class.

First, we set a variable `today` with today's date. The `Date.today` method creates an object that represents the current date. When the `Date.today` method is called, Ruby gets the current date from the computer's system clock.

Next we create a `birthday` variable and assign a new date that combines today's year with the month and day of the `birthdate`. This gives us the date of Foobar Kadigan's birthday this year.

The `Date` class can perform complex calendar arithmetic. The variables `birthdate` and `today` are *instances* of the `Date` class. We can use a greater-than operator to determine if Foobar Kadigan's birthday is in the future or the past.

The `if ...  else ...  end` structure is a *conditional statement*. If the birthday is in the future, we subtract `today` from `birthday` to calculate the number of days remaining until the owner's birthday, which we assign to the `countdown` variable.

If the birthday has already passed, we apply a `next_year` method to the birthday to get next year's birthday. Then we subtract `today` from `birthday.next_year` to calculate the number of days remaining until the owner's birthday, which we assign to the `countdown` variable.

The result might be fractional so we use the utility method `to_i` to convert the result to a whole number (integer) before assigning it to the `countdown` variable.

This shows you the power of programming in Ruby. Notice that I needed 16 paragraphs and over 600 words to explain 15 short lines of code. We used only

seven Ruby abstractions but they represent thousands of lines of code in the Ruby language implementation. With knowledge of Ruby syntax and the Ruby API, a few short lines of code in a text file gives us amazing ability.

In an upcoming chapter, we'll look more closely at the syntax and keywords of the Ruby language. But without knowing more than this, we can build a simple web application.

Let's see how we can put this functionality to use on a web page.

View

The Owner model provides the data we want to see on the Home page.

We'll create the markup and layout in a View file and add variables that present the data.

View files go in folders in the **app/views/** directory. In a typical application, one controller can render multiple views, so we make a folder to match each controller. You can make a new folder using your file browser or text editor. Or use the Unix **mkdir** command:

```
$ mkdir app/views/visitors
```

Create a file **app/views/visitors/new.html.erb**:

```erb
<h3>Home</h3>
<p>Welcome to the home of <%= @owner.name %>.</p>
<p>I was born on <%= @owner.birthdate %>.</p>
<p>Only <%= @owner.countdown %> days until my birthday!</p>
```

We've created a **visitors/** folder within the **app/views/** directory. We have only a single **new** view but if we had more views associated with the Visitors controller, they'd go in the **app/views/visitors/** folder.

We name our View file **new.html.erb**, adding the **.erb** file extension so that Rails will use the ERB templating engine to interpret the markup.

There are several syntaxes that can be used for a view file. In this tutorial, we'll use the ERB syntax that is most commonly used by beginners. Some experienced developers prefer to add gems that provide the Haml or Slim templating engines. As you might guess, a View that uses the Haml templating syntax would be named **new.html.haml**.

Our HTML markup is minimal, using only the `<h3>` and `<p>` tags. The only ERB markup we add are the `<%= ...  %>` delimiters. This markup allows us to insert Ruby code which will be replaced by the result of evaluating the code. In other words, `<%= @owner.name %>` will appear on the page as Foobar Kadigan.

You may have noticed that we refer to the Owner model with the variable `@owner`. It will be clear when we create the Visitors controller why we use this syntax (a variable name that begins with the `@` character is called an *instance variable*).

Obviously, if all we wanted to do was include the owner's name on the page, it would be easier to simply write the text. The Rails implementation becomes useful if the name is retrieved from a database or created programmatically.

We can better see the usefulness of the Owner model when we look at the use of `<= @owner.countdown %>`. There is no way to display a calculation using only static HTML, so Rails gives us a way to display the birthday countdown calculation.

If you're a programmer, you might wonder why we only output the variable on the page. Since we can use ERB to embed any Ruby code, we could perform the calculation right on the page by embedding `<%= (Date.new(today.year, @owner.birthdate.month, @owner.birthdate.day) - Date.today).to_i %>`. If you've used JavaScript or PHP, you may have performed calculations like this, right on the page. Rails would allow us to do so, but the practice violates the "separation of concerns" principle that encourages us to perform complex calculations in a model and only display data in the view.

Before we can display the home page, we need to create the Visitors controller.

Controller

The Visitors controller is the glue that binds the Owner model with the VisitorsController#new view.

Note: When we refer to a controller action, we use the notation "VisitorsController#new," joining the controller class name with the action (method) that renders a page. In this context, the **#** character is only a documentation convention.

Note: **VisitorsController** will be the class name and **visitors_controller.rb** will be the filename. The class name is written in camelCase (with a hump in the middle, like a camel) so we can combine two words without a space.

Unix commands get messy when filenames include spaces so we create a filename that combines two words with an underscore (sometimes called "snake_case").

Create a file **app/controllers/visitors_controller.rb**:

```ruby
class VisitorsController < ApplicationController

  def new
    @owner = Owner.new
  end

end
```

We define the class and name it **class VisitorsController**, inheriting behavior from the ApplicationController class which is defined in the Rails API.

We only need to define the **new** method. We create an *instance variable* named **@owner** and assign an instance of the Owner model. Any instance variables (variables named with the **@** character) will be available in the corresponding view file.

If we don't instantiate the Owner model, we'll get an error when the controller **new** action attempts to render the view because we use the **@owner** instance in the view file.

Keep in mind the purpose of the controller. Each controller action (method) responds to a request by obtaining a model (if data is needed) and rendering a view.

You've already created a view file in the **app/views/visitors** folder. The **new** action of the VisitorsController renders the template **app/views/visitors/new.html.erb**.

The **new** method is deceptively simple. Hidden behavior inherited from the ApplicationController does all the work of rendering the view. We can make the hidden code explicit if we wish to. It would look something like this:

```ruby
class VisitorsController < ApplicationController

  def new

    @owner = Owner.new
    render 'visitors/new'
  end

end
```

This is an example of Rails magic. Some developers complain this is black magic because the "convention over configuration" principle leads to obscurity. Rails often offers default behavior that looks like magic because the underlying implementation is hidden in the depths of the Rails code library. This can be frustrating when, as a beginner, you want to understand what's going on.

Revealing the hidden code, we see that invoking the **new** method calls a **render** method supplied by the ApplicationController parent class. The **render** method searches in the **app/views/visitors** directory for a view file named **new** (the file extension **.html.erb** is assumed by default). The code underlying the **render** method is complex. Fortunately, all we need to do is define the method and instantiate the Owner model. Rails takes care of the rest.

As a beginner, simply accept the magic and don't confound yourself trying to

find how it works. As you gain experience, you can dive into the Rails source code to unravel the magic.

Scaffolding

This tutorial aims to give you a solid foundation in basic concepts. The model–view–controller pattern is one of the most important. I've found the best way to understand model–view–controller architecture is to create and examine the model, view, and controller files.

As you continue your study of Rails, you'll find other tutorials that use the *scaffolding* shortcut. For example, Rails Guides: Getting Started with Rails includes a section "Getting Up and Running Quickly with Scaffolding" which shows how to use the `rails generate scaffold` command to create model, view, and controller files in a single operation. Students often use scaffolding to create simple Rails applications.

In practice, I've observed that working Rails developers seldom use scaffolding. There's nothing wrong with it; it just seems that scaffolding doesn't offer much that can't be done as quickly by hand.

Test the Application

We've created a model, view, and controller. Now let's run the application.

Enter the command:

```
$ rails server
```

Open a web browser window and navigate to http://localhost:3000/. You'll see our new home page.
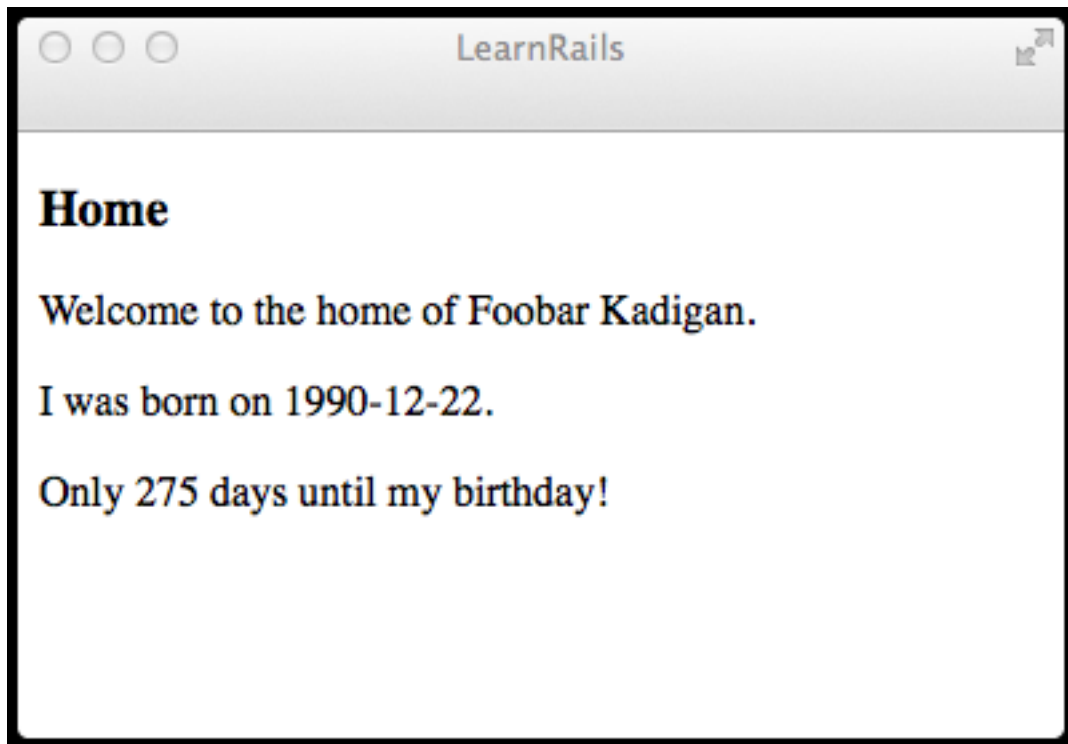
Figure 13.1: Dynamic home page shows days until a birthday.

It's a very simple web page but it uses Ruby to calculate the countdown to the birthday. And the underlying code conforms to the conventions and structure of Rails.

Git

At this point, you might have the Rails server running in your console window. We're going to run a git command in the console now.

You might think you have to enter Control-c to shut down the server and get the command prompt. But that's not necessary. You can open more than one console window. Your terminal application lets you open multiple tabs so you can easily switch between windows without using a lot of screen real estate.

If you haven't tried it, now is a good time. It is convenient to have a console window open for the server and another for various Unix commands.

Let's commit our changes to the Git repository and push to GitHub:

```
$ git add -A
$ git commit -m "dynamic home page"
$ git push
```

Now let's take a look at troubleshooting.

Chapter 14

Troubleshoot

In the last chapter, we built a dynamic home page and learned about the model–view–controller architecture of Rails. There was a lot to learn, but the code was simple, and I hope it worked the first time you tried it.

Before we do any more work on our tutorial application, we need to learn about troubleshooting and debugging. As a software developer, you'll spend a lot of time with code that doesn't work. You'll need tools and techniques to diagnose problems.

Git

In this chapter we'll make changes to the application just for troubleshooting.

Before you get started, make sure the work you've done is committed to your git repository. Use the `git status` command to check:

```
$ git status
```

You should see:

```
On branch master
nothing to commit, working directory clean
```

If **git status** reports any uncommitted changes, go back to the last step in the previous chapter and commit your work to the git repository before continuing. At the end of this chapter, we're going to throw away the work we've done in this chapter. We don't want to accidentally throw away work from the previous chapter so make sure it is committed to the repository.

Interactive Ruby Shell

There will be times when you want to try a snippet of Ruby code just to see if it works. Your tool will be IRB, the Interactive Ruby shell.

IRB is a Ruby interpreter that runs from the command line. It executes any Ruby code and provides an immediate response, allowing you to experiment in real-time.

Let's try it.

```
$ irb
>>
```

The command **irb** launches the program and displays a prompt that shows your Ruby version, a line number, and an arrow. I'll just show a simple prompt in the examples, instead.

If you enter a valid Ruby expression, the interpreter will display the result of evaluating the expression.

Try simple arithmetic:

```
>> n = 2
 => 2
>> n + 2
 => 4
```

Wow! You are using your computer for simple math. Maybe you can delete the calculator app from your phone.

IRB will evaluate any Ruby expression and helps you quickly determine if syntax and logic is correct.

IRB for Blocks of Code

At first glance, it appears IRB works on just one line of code.

Actually, IRB can handle multiple lines of code. Try it:

```
>> n = 10
=> 10
>> if n < 10
>>   puts "small"
>> else
?>   puts "big"
>> end
big
=> nil
>>
```

Here we set `n = 10` and then enter a conditional statement line-by-line. After we enter the final `end`, IRB interprets the code and outputs the result.

You'll often enter more than one line of code in IRB. If you find yourself frustrated because you've entered typos and had to enter the same code repeatedly, you can use IRB to load code you've saved in a file:

```
>> load './mytest.rb'
```

Quitting IRB

It can be very frustrating to find you are stuck inside IRB. Unlike most shell commands, you can't quit with Control-c. Enter Control-d or type **exit** to quit IRB:

```
$ irb
>> exit
```

Learn More About IRB

Here's an entertaining way to learn about IRB:

- Why's (Poignant) Guide to Ruby (with a Basic Introduction to IRB)

Here's a more conventional way to learn about IRB:

- The Pragmatic Programmer's Guide

Beyond IRB

If you ask experienced Rails developers for help with IRB, they'll often recommend you switch to Pry. Pry is a powerful alternative to the standard IRB shell for Ruby. As you gain experience, you might take a look at Pry to see what the enthusiasm is all about. But for now, as a beginner trying out a few lines of Ruby code, there's no need to learn Pry.

Rails Console

IRB only evaluates expressions that are defined in the Ruby API. IRB doesn't know Rails.

It'd be great to have a tool like IRB that evaluates any expression defined in the Rails API. The tool exists; it's called the Rails console. It is particularly useful because it loads your entire Rails application. Your application will be running as if the application was waiting to respond to a web request. Then you can expose behavior of any pieces of the web application.

```
$ rails console
...
Loading development environment (Rails 5.x.x)
>>
```

The Rails console behaves like IRB but loads your Rails development environment. The prompt shows it is ready to evaluate an expression.

Let's use the Rails console to examine our Owner model:

```
>> myboss = Owner.new
 => #<Owner:0x007fc18e91faf8>
```

We've created a variable named **myboss** and created a new instance of the Owner class. The Rails console responds by displaying the unique identifier it uses to track the object. The identifier is not particularly useful, except to show that something was created.

If you're unsure about the difference between an *instance* and a *class*, we've just seen that we can make one or more instances of a class by calling the **Owner.new** method. When we specify the **Owner** class, the class definition is loaded into the computer's working memory (our development environment) from the class definition file on disk. Then we can use the **Owner.new** method to make one or more instances of the **Owner** class. Each instance is a unique

object with its own data attributes but the same behavior as other objects in-stantiated from its class.

Let's assign the name of our boss to a variable called **name**:

```
>> name = myboss.name
 => "Foobar Kadigan"
```

Our variable **myboss** is an instance of an **Owner** class so it responds to the method **Owner.name** by returning the owner's name.

We want to show respect to our boss so we'll perform some *string manipulation*:

```
>> name = 'Mr. ' + name
 => "Mr. Foobar Kadigan"
```

We're done for now. When we quit the Rails console or shut down the computer the **Owner** class definition remains stored on disk but the instances disappear. The bits that were organized to create the variable **name** will evaporate into the ether.

Actually, the bits are still there, in the form of logic states in the computer's chips, but they have no meaning until another program uses them.

Enter Control-d or type **exit** to quit the Rails console.

The Rails console is a useful utility. It is like a handy calculator for your code. Use it when you need to experiment or try out short code snippets.

Rails Logger

As you know, a Rails application sends output to the browser that makes a web request. On every request, it also sends diagnostic output to the *server*

log file. Depending on whether the application is running in the development environment or in production, the log file is here:

- **log/development.log**

- **log/production.log**

In development, everything written to the log file appears in the console window after you run the **rails server** command. Scrolling the console window is a good way to see diagnostics for every request.

Here's what you see in the log after you visit the application home page:

```
Started GET "/" for ::1 at ...
Processing by VisitorsController#new as HTML
  Rendering visitors/new.html.erb within layouts/application
  Rendered visitors/new.html.erb within layouts/application (6.6ms)
Completed 200 OK in 650ms (Views: 634.4ms | ActiveRecord: 0.0ms)
```

You may have more than one console window open in the terminal application. If you don't see your log output in your terminal, check if you have tabs with other windows.

Here's the best part. You can add your own messages to the log output by using the Rails logger. Let's try it out.

Modify the file **app/controllers/visitors_controller.rb**:

```
class VisitorsController < ApplicationController

  def new
    Rails.logger.debug 'DEBUG: entering new method'
    @owner = Owner.new
    Rails.logger.debug 'DEBUG: Owner name is ' + @owner.name
  end

end
```

Visit the home page again and you'll see this in the console output:

```
Started GET "/" for ::1 at ...
Processing by VisitorsController#new as HTML
DEBUG: entering new method
DEBUG: Owner name is Foobar Kadigan
  Rendering visitors/new.html.erb within layouts/application
  Rendered visitors/new.html.erb within layouts/application (1.1ms)
Completed 200 OK in 81ms (Views: 72.2ms | ActiveRecord: 0.0ms)
```

If you really needed to do so, you could add a logger statement at every step in the application. You could see how the application behaves, step by step. And you could "print" the value of every variable at every step. You'll never need diagnostics at this level of detail in Rails, but the logger is extremely useful when you are trying to understand unexpected behavior.

Let's add logger statements to the **Owner** model. Modify the file **app/models/owner.rb**:

```ruby
class Owner

  def name
    name = 'Foobar Kadigan'
  end

  def birthdate
    birthdate = Date.new(1990, 12, 22)
  end

  def countdown
    Rails.logger.debug 'DEBUG: entering Owner countdown method'
    today = Date.today
    birthday = Date.new(today.year, birthdate.month, birthdate.day)
    if birthday > today
      countdown = (birthday - today).to_i
    else
      countdown = (birthday.next_year - today).to_i
    end
  end

end
```

We added the **Rails.logger.debug** statement to the **Owner.countdown** method.

Visit the home page and here's what you'll see in the console output:

```
Started GET "/" for ::1 at ...
Processing by VisitorsController#new as HTML
DEBUG: entering new method
DEBUG: Owner name is Foobar Kadigan
  Rendering visitors/new.html.erb within layouts/application
DEBUG: entering Owner countdown method
  Rendered visitors/new.html.erb within layouts/application (0.7ms)
Completed 200 OK in 69ms (Views: 61.1ms | ActiveRecord: 0.0ms)
```

You'll often need to "get inside" the model or controller to see what's happening. The Rails logger is the best tool for the job.

Here are some tricks for the Rails logger.

In a controller, you can use the method **logger** on its own. In a model, you have to write **Rails.logger** (both class and method).

You can use any of the methods **logger.debug**, **logger.info**, **logger.warn**, **logger.error**, or **logger.fatal** to write log messages. By default, you'll see any of these messages in the development log. Log messages written with the **logger.debug** method will not be recorded in a production log file.

If you want your log messages to stand out, you can add formating code for color:

```
Rails.logger.debug "\033[1;34;40m[DEBUG]\033[0m " + 'will appear in bold blue'
```

For more about the Rails logger, see the RailsGuide: Debugging Rails Applications.

Revisiting the Request-Response Cycle

Earlier, when we investigated the request-response cycle, we looked in the server log to see the response to the web browser request.

Now, with debug statements in the controller and model, we'll see messages showing the server's traverse of the model-view-controller architecture.

```
Started GET "/" for ::1 at ...
Processing by VisitorsController#new as HTML
DEBUG: entering new method
DEBUG: Owner name is Foobar Kadigan
  Rendering visitors/new.html.erb within layouts/application
DEBUG: entering Owner countdown method
  Rendered visitors/new.html.erb within layouts/application (0.7ms)
Completed 200 OK in 69ms (Views: 61.1ms | ActiveRecord: 0.0ms)
```

Notice how the diagnostic messages in the console window match the headers in the browser diagnostic view. The browser's "Request Method:GET" matches the server's "Started GET." The browser's "Status Code: 200" matches the server's "Completed 200 OK" (you might have to clear the browser's cache if the browser is showing "304 Not Modified").

We can see evidence of the model-view-controller architecture. "Processing by VisitorsController#new" shows the program flow entering the controller. Our debug statements show we enter the **new** method and reveal the value of the Owner name. The next debug statement reveals the flow has passed to the Owner model. A diagnostic message shows the controller has rendered the **visitors/new.html.erb** view file. Finally, the "Completed 200 OK" message indicates the response has been sent to the browser.

As we learned, the model-view-controller architecture is an abstract design pattern. We've seen it reflected in the file structure of the Rails application directory. Now we can see it as activity in the server log.

The Stack Trace

The Rails logger is extremely useful if you want to insert messages to show program flow or display variables. But there will be times when program flow halts and the console displays a *stack trace*.

Let's deliberately create an error condition and see an error page and stack trace.

Modify the file **app/controllers/visitors_controller.rb**:

```ruby
class VisitorsController < ApplicationController

  def new
    Rails.logger.debug 'DEBUG: entering new method'
    @owner = Owner.new
    Rails.logger.debug 'DEBUG: Owner name is ' + @owner.name
    DISASTER
  end

end
```

Visit the home page and you'll see an error page:

You'll see this error page because we've installed the better_errors gem. Without the better_errors gem, you'd see the default Rails error page which is quite similar.

In the console log, the stack trace will show everything that happens before Rails encounters the error:

```
Started GET "/" for ::1 at ...
Processing by VisitorsController#new as HTML
DEBUG: entering new method
DEBUG: Owner name is Foobar Kadigan
Completed 500 Internal Server Error in 8ms (ActiveRecord: 0.0ms)

NameError – uninitialized constant VisitorsController::DISASTER:
  app/controllers/visitors_controller.rb:7:in `new'
  .
  .
  .
```

To save space, I'm only showing the top line of the stack trace. I've eliminated about sixty lines from the stack trace.

Don't feel bad if your reaction to a stack trace is an immediate, "TMI!" Indeed, it is usually Too Much Information. There are times when it pays to carefully
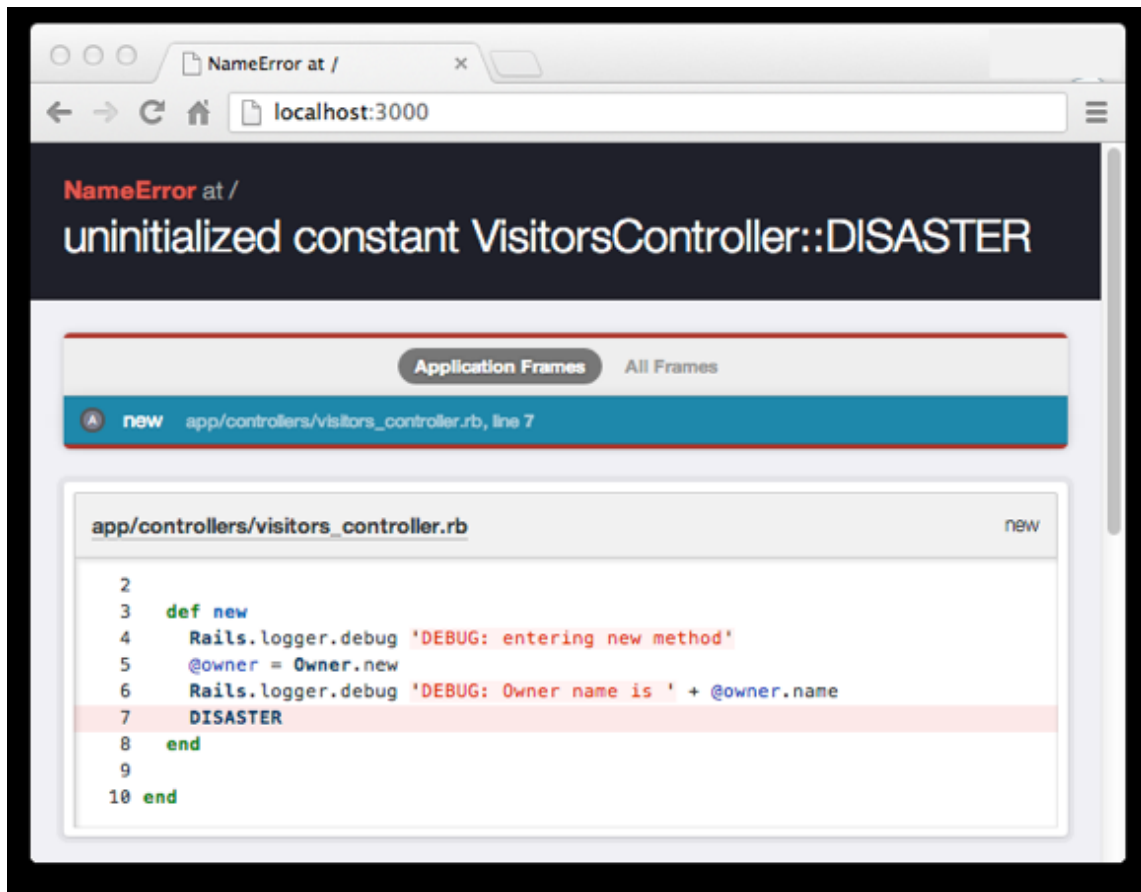
Figure 14.1: Error page.

read through the stack trace line by line, but most often, only the top line of the stack trace is important.

In this case, both the error page and the top line of the stack trace show the application failed (I prefer to say, "barfed") when it encountered an "uninitialized constant" at line 7 of the **app/controllers/visitors_controller.rb** file in the **new** method. It's easy to find line 7 in the file and see that is exactly where we added a string that Rails doesn't understand.

The point of this exercise is to encourage you to read the top line of the stack trace and use it to diagnose the problem. I'm always surprised how many de-

velopers ignore the stack trace, probably because it looks intimidating.

Raising an Exception

As you just saw, you can purposefully break your application by adding characters that Rails doesn't understand. However, there is a better way to force your program to halt, called *raising an exception*.

Let's try it. Modify the file **app/controllers/visitors_controller.rb**:

```ruby
class VisitorsController < ApplicationController

  def new
    Rails.logger.debug 'DEBUG: entering new method'
    @owner = Owner.new
    Rails.logger.debug 'DEBUG: Owner name is ' + @owner.name
    raise 'Deliberate Failure'
  end

end
```

You can throw an error by using the **raise** keyword from the Ruby API. You can provide any error message you'd like in quotes following **raise**.

Here's the console log after you try to visit the home page:

```
Started GET "/" for ::1 at ...
Processing by VisitorsController#new as HTML
DEBUG: entering new method
DEBUG: Owner name is Foobar Kadigan
Completed 500 Internal Server Error in 6ms (ActiveRecord: 0.0ms)

RuntimeError - Deliberate Failure:
  app/controllers/visitors_controller.rb:7:in `new'
  .
  .
  .
```

Before we continue, let's remove the deliberate failure. Modify the file **app/controllers/visit**

```
class VisitorsController < ApplicationController

  def new
    Rails.logger.debug 'DEBUG: entering new method'
    @owner = Owner.new
    Rails.logger.debug 'DEBUG: Owner name is ' + @owner.name
  end

end
```

Rails and the Ruby API provide a rich library of classes and methods to raise and handle exceptions. For example, you might want to display an error if a user enters a birthdate that is not in the past. Rails includes various exception handlers to display errors in production so users will see a helpful web page explaining the error.

Git

There's no need to save any of the changes we made for troubleshooting.

You could go to each file and carefully remove the debugging code you added. But there's an easier way.

Check which files have changed:

```
$ git status
# Changes not staged for commit:
#   (use "git add ..." to update what will be committed)
#   (use "git checkout -- ..." to discard changes in working directory)
#
# modified:   app/controllers/visitors_controller.rb
# modified:   app/models/owner.rb
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Use Git to revert your project to the most recent commit:

```
$ git reset --hard HEAD
```

The Git command `git reset -hard HEAD` discards any changes you've made since the most recent commit. Check the status to make sure:

```
$ git status
# On branch master
nothing to commit, working directory clean
```

We've cleaned up after our troubleshooting exercise.

Chapter 15

Just Enough Ruby

Experienced Rails developers debate whether beginners should study Ruby before learning Rails.

By all means, if you love the precision and order of programming languages, dive into the study of Ruby from the beginning. But most people don't delay starting Rails while learning Ruby; realistically, you'll retain more knowledge of Ruby if you learn it as you build things in Rails. That is the approach we've taken in this book. You've already built a simple Rails application and used Ruby as you did so.

Reading Knowledge of Ruby

What you need, more than anything, when you start working with Rails, is reading knowledge of Ruby.

With a reading knowledge of Ruby you'll avoid feeling overwhelmed or lost when you encounter code examples or work through a tutorial. Later, as you tackle complex projects and write original code, you'll need to know enough of the Ruby language to implement the features you need. But as a student, you'll be following tutorials that give you all the Ruby you need. Your job is

to recognize the language keywords and use the correct syntax when you type Ruby code in your text editor.

To that end, this chapter will review the Ruby keywords and syntax you've already learned. And you'll extend your knowledge so you'll be prepared for the Ruby you'll encounter in upcoming chapters.

Ruby Example

To improve your reading knowledge of Ruby, we'll work with an example file that contains a variety of Ruby expressions.

We won't use this file in our tutorial application, so you'll delete it at the end of this chapter. But we'll approach it as real Ruby code, so make a file and copy the code using your text editor.

First we have to consider where the file should go. It will not be a model, view, controller, or any other standard component of Rails. Rails has a place for miscellaneous files that don't fit in the Rails API. We'll create the file in the **lib/** folder. That's the folder you'll use for any supporting Ruby code that doesn't fit elsewhere in the Rails framework.

Create a file **lib/example.rb**:

```ruby
class Example < Object

  # This is a comment.

  attr_accessor :honorific
  attr_accessor :name
  attr_accessor :date

  def initialize(name,date)
    @name = name
    @date = date.nil? ? Date.today : date
  end

  def backwards_name
    @name.reverse
  end
```

```ruby
  def to_s
    @name
  end

  def titled_name
    @honorific ||= 'Esteemed'
    titled_name = "#{@honorific} #{@name}"
  end

  def december_birthdays
    born_in_december = [ ]
    famous_birthdays.each do |name, date|
      if date.month == 12
        born_in_december << name
      end
    end
    born_in_december
  end

  private

  def famous_birthdays
    birthdays = {
      'Ludwig van Beethoven' => Date.new(1770,12,16),
      'Dave Brubeck' => Date.new(1920,12,6),
      'Buddy Holly' => Date.new(1936,9,7),
      'Keith Richards' => Date.new(1943,12,18)
    }
  end

end
```

In some ways, this Ruby code is like a poem from Lewis Carroll:

```
'Twas brillig, and the slithy toves
Did gyre and gimble in the wabe;
All mimsy were the borogoves,
And the mome raths outgrabe.

"Beware the Jabberwock, my son!
The jaws that bite, the claws that catch!
Beware the Jubjub bird, and shun
The frumious Bandersnatch!"
```

The poem corresponds to the rules of English syntax but is nonsense.

The code follows the rules of Ruby syntax, and unlike the poem, uses meaningful words. But it is unclear how the author intends anyone to use the code. If you're beginning a career as a Rails developer, this won't be the last time you look at code and wonder what the author was intending. In this case, I just want to give you some code that illustrates typical Ruby syntax and structure.

Ruby Keywords

When reading Ruby code, the first challenge is determining which words are Ruby keywords and which were made up by the developer. Code is only strings of characters. But some strings have special meaning for everyone and all others are arbitrary words that only have meaning to an individual developer.

As you gain experience, you'll recognize Ruby keywords because you've seen them before.

You'll also recognize a developer's made-up words because of their position relative to other words and symbols. Some made-up words will be obvious because they are just too idiosyncratic to be part of the Ruby language. For example, you'll rightly guess that **myapp** or **fluffycat** are not part of the Ruby language.

If you're reading a Lewis Carroll poem, you could look up words in a dictionary to see if you find them.

There is only one way to be sure which words are part of the Ruby language: Check the Ruby API.

As an exercise, pick one of the words from the example code that you think might be a Ruby keyword and search the API to find it.

If you want to be a diligent student, you can check every keyword in the example code to find out whether it is in the Ruby API. It is more practical to learn to recognize Ruby keywords, which we'll do next.

API Documentation

The Ruby API documentation lists every keyword in the language:

- ruby-doc.org - the official Ruby API

- apidock.com/ruby - Ruby API docs with usage notes

Ruby Files

When we write code, we save it in files. We've added our miscellaneous example file to the **lib/** folder.

By convention, Ruby files end with the file extension **.rb**.

Using IRB

In the "Troubleshooting" chapter, you used IRB (the Interactive Ruby Shell) to try out Ruby code. You can use IRB to try out the example code in the console.

```
$ irb
>> load 'lib/example.rb'
=> true
>>  require 'date'
=> true
>> ex = Example.new('Daniel',nil)
=> #<Example:0x007fb46c9eecd8 @name="Daniel", @date=#<Date: 2015-12-23 ...
>> list = ex.december_birthdays
=> ["Ludwig van Beethoven", "Dave Brubeck", "Keith Richards"]
>>
```

Entering the **load** directive and the filename brings the code into IRB.

The **require 'date'** statement loads the Ruby date library.

The statement `ex = Example.new('Daniel',nil)` creates an object from the `Example` class.

The `ex.december_birthdays` method returns an array of names.

Remember you can use Control-d to exit from IRB.

Now, for practice, we'll read the Ruby code.

Whitespace and Line Endings

Whitespace characters such as spaces and tabs are generally ignored in Ruby code, except when they are included in strings. There are several special cases where whitespace is significant in Ruby expressions but you are not likely to encounter these cases as a beginning Rails developer.

Some programming languages (Java and the C family) require a semicolon as a terminator character at the end of statements. Ruby does not require a semicolon to terminate a statement. Instead, if the Ruby code on a line is a complete expression, the line ending signifies the end of the statement. If the line ends with a `+` or other operator, or a backslash character, the statement is split into multiple lines.

Comments

Ruby ignores everything that is marked as a comment. Use comments for notes to yourself or other programmers.

```
# This is a comment.
```

You can also turn code into comments if you don't want the code to run. This is a common trick when you want to "turn off" some part of your code but you don't want to delete it just yet, because you are trying out alternatives.

The Heart of Programming

Three principles are at the heart of all programming:

- syntax

- conditional execution

- transformation

Computers allow no ambiguity. Code must exactly follow the *syntax* of a language. Typos, guesses, and code that is almost-but-not-quite right will simply fail, often without any helpful error messages.

Computers seem intelligent because they can execute code *conditionally*. You can write a program so that given one set of conditions, certain parts of the code will execute, and given different conditions, other parts of the code will execute.

Lastly, programs are written to transform abstractions from one form to another. That's why computer programs look like math. When we learn simple arithmetic, we learn we can take the symbols for numbers and add them together to make a different number. Computer programs do more than add numbers; a program can transform words and other abstractions.

Assignment

In Ruby, like many other programming languages, the equals sign indicates we are *assigning* a value.

```
name = 'Foobar Kadigan'
```

Assignment is the first step to transformation. Here **`'Foobar Kadigan'`** is a string of letters. The equals sign is the assignment operator. And **`name`** is a *variable* that stores the value so it can be easily reused. We don't have to type **`'Foobar Kadigan'`** every time we need a name; we can use **`name`** instead.

Just as we can assign a value to a variable, we can reassign a new value whenever we want. That is why we call them variables; the value can vary.

```
name = 'Mr. Foobar Kadigan'
```

Variables can be assigned strings of letters, numbers, or anything else. "Anything else" is very broad because we can use Ruby to make complex structures that contain data and also "do work." These complex structures are *objects* and we say that Ruby is *object-oriented* because it is easy to work with objects in Ruby.

Object-Oriented Terminology

Software architects use a common vocabulary to talk about programming languages:

- **class**

- **instance** or **object**

- **method**

- **attribute** or **property**

- **inheritance**

- **class hierarchy**

There are three ways to learn what these words mean. You can memorize the definitions. You can write code and intuitively grasp the meanings. Or you can gain an understanding by applying metaphors.

Houses

For example, some programming textbooks attempt to explain a *class* like this: A blueprint for a house design is like a *class definition*. All the houses built from that blueprint are *objects* of a class we could call House.

Vehicles

Or: The concept of "vehicle" is like a *class*. Vehicles can have *attributes*, like color or number of doors. They have behavior, or *methods*, like buttons that turn on lights or honk a horn. The concepts of "truck" or "car" are also classes, *inheriting* common characteristics from a *superclass* "vehicle." The blue car in your driveway with four doors is an object, a particular *instance* of the class "car."

Cookies

I like the cookie metaphor the best.

A *class definition* is like a cookie cutter.

Bits in the computer memory are like cookie dough.

The cookie cutter makes as many individual cookies as you want. Each cookie is an *instance* of the Cookie class, with the same shape and size as the others. Cookies are *objects*.

You can decorate each cookie with sprinkles, which are *attributes* that are unique to each instance. Some cookies get red sprinkles, some get green, but

the shape remains the same.

Running a program is like baking. The cookies change state from raw to cooked.

Sticking a toothpick in a cookie is like calling a *method*. The method returns a result that tells you about the state: Is it done?

Limitations of Metaphors

Metaphors are imperfect.

If baking was like running a program, all the cookies would disappear as soon as the oven was turned off.

When a software program contains a "car" model, it doesn't fully model cars in the physical world. It represents an abstraction of characteristics a programmer deems significant. Let's make a model for a Person that contains an attribute Gender. What values are possible for the attribute Gender? For many years, Facebook offered two choices, male and female. In 2014, Facebook suddenly offered a choice of over fifty gender terms. As Sarah Mei discusses in a blog post, Why Gender is a Text Field on Diaspora, your assumptions have consequences when you build a model.

Most classes in software APIs don't model anything in the real world. They typically represent an abstraction, like an Array or a Hash, which inherits characteristics from another abstraction, for example, a Collection.

Given the limitations of metaphors, maybe it is easier to simply say that software allows us to create abstractions that are "made real" and then manipulated and transformed. Terminology such as *class* and *instance* describe the abstractions and the relationships among them.

Definitions

Here are definitions for some of the terms we encounter when we consider Rails from the perspective of a software architect:

class an abstraction that encapsulates data and behavior

class definition written code that describes a class

instance or object a unique copy of a class that exists only while a program is running

inheritance a way to make a class by borrowing from another class

class hierarchy classes that are related by inheritance

method a command that returns data from an object

attribute or property data that can be set or retrieved from the object

variable a name that can be assigned a value or object

expression or statement any combination of variables, classes, and methods that returns a result

Some of these terms are abstractions that are "made real" in the Ruby API (such as class and method); others are just terms that describe code, much like we use terms such as "adjective" or "noun" to talk about the grammar of the English spoken language.

Classes

You don't have to create classes to program in Ruby. If you only write simple programs, you won't need classes. Classes are used to organize your code and

make your software more modular. For the software architect, classes make it possible to create a structure for complex software programs. To use Rails, you'll use the classes and methods that are defined in the Rails API.

There is one class at the apex of the Ruby class hierarchy: **BasicObject**. **BasicObject** is a very simple class, with almost no methods of its own. The **Object** class inherits from **BasicObject**. All classes in the Ruby and Rails APIs inherit behavior from **Object**. **Object** provides basic methods such as **nil?** and **to_s** ("to string") for every class that inherits from **Object**.

We create a class **Example** and inherit from **Object** with the **<** "inherits from" operator:

```
class Example < Object
.
.
.
end
```

The **end** statement indicates all the preceding code is part of the **Example** class.

In Ruby, all classes inherit from the Object class, so we don't need to explicitly *subclass* from **Object** as we do here. The example just shows it for teaching purposes.

Here is the **Example** class without the explicit subclassing from **Object**:

```
class Example
.
.
.
end
```

Much of the art of programming is knowing what classes are available in the API and deciding when to subclass to inherit useful methods.

Methods

Classes give organization and structure to a program. Methods get the work done.

Any class can have methods. Methods are a series of expressions that return a result (a value). We say methods describe the class behavior.

A method definition begins with the keyword **def** and (predictably) ends with **end**.

```
def backwards_name
  @name.reverse
end
```

Initializing the object and calling the method returns a result:

```
ex = Example.new('Daniel',nil)
my_backwards_name = ex.backwards_name
  => leinaD
```

We can also *override* a method from the parent class.

```
def to_s
  @name
end
```

Here we are *overriding* the **to_s** ("to string") method from the parent **Object** class.

Ordinarily, the **to_s** method returns the object's class name and an object id. Here we will return the string assigned to the variable **@name**.

Most times you won't override the **to_s** ("to string") method. This example shows how you can override any method inherited from a parent class.

Dot Operator

The "dot" is the method operator. This tiny punctuation symbol is a powerful operator in Ruby.

It allows us to *call a method* to get a result.

Sometimes we say we *send a message* to the object when we invoke a method, implying the object will send a result.

Some classes, such as **Date**, provide *class methods* which can be called directly on the class without instantiating it first. For example, you can run this in the Rails console:

```
Date.today
 => Tue, 15 Oct 2013
```

More often, methods are called on variables which are instances of a class. For example:

```
birthdate = Date.new(1990, 12, 22)
 => Sat, 22 Dec 1990
birthmonth = birthdate.month
 => 12
```

We can apply *method chaining* to objects. For example, **String** has methods **reverse** and **upcase** (among many others). We could write:

```
nonsense = 'foobar'
 => "foobar"
reversed = nonsense.reverse
 => "raboof"
capitalized = reversed.upcase
 => "RABOOF"
```

It is easier to use method chaining and write:

```
'foobar'.reverse.upcase
 => "RABOOF"
```

Classes create a structure for our software programs and methods do all the work.

Question and Exclamation Methods

You'll see question marks and exclamation points (sometimes called the "bang" character) used in method names. These characters are simply a naming convention for Ruby methods.

The question mark indicates the method will return a *boolean value* (true or false).

The bang character indicates the method is "dangerous." In some cases it means the method will change the object rather just return a result. In Rails an exclamation point often means the method will throw an exception on failure rather than failing silently.

Initialize Method

Objects are created from classes before they are used. As I suggested earlier, class definitions are cookie cutters; the Ruby interpreter uses them to cut cookies. When we call the `new` method, we press the cookie cutter into the dough and get a new object. All the cookies will have the same shape but they can be decorated differently, by sprinkling attributes of different values.

The `initialize` method is one of the ways we sprinkle attributes on our cookie.

```ruby
def initialize(name,date)
```

When we want to use an **Example** object and assign it to a variable, we will instantiate it with **Example.new(name,date)**. The **new** method calls the **initialize** method automatically. If we don't define an **initialize** method, the **new** method still works, inherited from **Object**, so we can always instantiate any class.

Method Parameters

Methods are useful when they operate on data.

If we want to send data to a method, we define the method and indicate it will accept *parameters*. Parameters are placeholders for data values. The values that are passed to a method are *arguments*. "Parameters" are empty placeholders and "arguments" are the actual values. In practice, "parameters" and "arguments" are terms that are used interchangeably and not many developers will notice if you mix up the terms.

Our **initialize** method takes **name** and **date** arguments:

```ruby
def initialize(name,date)
```

Ruby is clever with method parameters. You can define a method and specify default values for parameters. You can also pass extra arguments to a method if you define a method that allows optional parameters. This makes methods very flexible.

We separate our parameters with commas. For readability, we enclose our list of parameters in parentheses. In Ruby, parentheses are always optional but they often improve readability.

Variable

In Ruby, everything is an object. We can assign any object to a variable. The variable works like an alias. We can use a variable anywhere as if it were the assigned object. The variable can be assigned a string, a numeric value, or an instance of any class (all are objects).

```
name
```

You can assign a new value to a variable anywhere in your method. You can assign a different kind of object if you want. You can take away someone's name and give them a number. We can create a variable **player**, assign it the string **'Jackie Robinson'**, replace the value with an integer **42**, or even a date such as **Date.new(1947,4,15)**.

Symbol

Obviously, we see many symbols when we read Ruby code, such as punctuation marks and alphanumeric characters. But *symbol* has a specific meaning in Ruby. It is like a variable, but it can only be assigned a value once. After the initial assignment, it is *immutable*; it cannot be changed.

You will recognize a symbol by the colon that is always the first character.

```
:name
```

Symbols are efficient and fast because the Ruby interpreter doesn't have to work to check their current values.

You'll often see symbols used in Rails where you might expect a variable.

Attributes

In an object, methods do the work and data is stored as variables. We can use the **initialize** method to input data to the object. We can't access data in variables from outside the object unless it is exposed as *attributes*.

Classes can have attributes, which we can "set" and "get." That is, we can establish a value for an attribute and retrieve the value by specifying the attribute name.

Attributes are a convenient way to push data to an object and pull it out later.

In Ruby, attributes are also called properties.

Here we use the **attr_accessor** directive to specify that we want to expose **honorific**, **name** and **date** attributes.

```ruby
attr_accessor :honorific
attr_accessor :name
attr_accessor :date
```

If we use **attr_accessor** to establish attributes, we can use the attribute names as methods. For example, we could write:

```ruby
ex = Example.new('Daniel',nil)
my_name = ex.name
```

In Ruby, attributes are just specialized methods that expose data outside the object.

Instance Variable

Inside an object, an ordinary variable only can be used within the method in which it appears. If you use a variable with the same name in two different

methods, it will have a different value in each method. The *scope* of a variable is limited to the method in which it is used.

Often you want a variable to be available throughout an instance, within any method. You can declare an *instance variable* by using an `@` (at) sign as the first character of the variable name.

The instance variable can be used by any method after the class is instantiated.

```
@name = name
```

The values assigned to instance variables are unique for every instance of the class. If you create multiple instances of a class, each will have its own values for its instance variables. Here we create two instances of the **Example** class. The **@name** instance variable will be "Daniel" in the first instance and "Foobar" in the second instance.

```
ex1 = Example.new('Daniel',nil)
ex2 = Example.new('Foobar',nil)
```

An instance variable is not visible outside the object in which it appears; but when you create an **attr_accessor**, it creates an instance variable and makes it accessible outside the object.

Instance Variables in Rails

In a Rails controller, you'll often see a model assigned to an instance variable. Earlier we saw **@owner = Owner.new** when we instantiated an Owner model. We use an instance variable when we want a model to be available to the view template.

Rails beginners learn the simple rule that you have to use the `@` (at) sign if you want a variable to be available in the view. Intermediate Rails developers

learn that the variable with the `@` (at) sign is called an instance variable and is only available within the *scope* of the instance (practically speaking, to other methods in the class definition). That leads to a question: Why is an instance variable available inside a view?

There is a good reason. A Rails view is NOT a separate class. It is a template and, under the hood, it is part of the current controller object. From the viewpoint of a programmer, a Rails controller and a view are separate files, segregated in separate folders. From the viewpoint of a software architect, the controller is a single object that evaluates the template code, so an instance variable can be used in the view file.

This example shows us that the programmer and the software architect have different perspectives on a Rails application. Understanding Rails requires an integration of multiple points of view.

Double Bar Equals Operator

I've suggested that the best way to get help is to use Google or Stack Overflow to look for answers. But that's difficult when you don't know what symbols are called. Try googling "‖=" and you'll get no results. Instead, try googling "bar bar equals ruby" or "double pipe equals ruby" and you'll find many explanations of the "or equals" operator. This is an example of mysterious shorthand code you'll often find in Rails.

"‖=" is used for conditional assignment. In this case, we only assign a value to the variable if no value has been previously assigned.

```
@honorific ||= 'Esteemed'
```

It is equivalent to this conditional expression:

```
if not x
  x = y
end
```

Conditional assignment is often used to assign a "default value" when no other value has been assigned.

Conditional

Conditional logic is fundamental to programming. Our code is always a path with many branches.

When the Ruby interpreter encounters an **if** keyword, it expects to find an expression which evaluates as true or false (a *boolean*).

If the expression is true, the statements following the condition are executed.

If the expression is false, any statements are ignored, unless there is an **else**, in which case an alternative is executed.

```
if date.month == 12

  .

  .

  .
end
```

Sometimes you'll see **unless** instead of **if**, which is a convenient way of saying "execute the following if the condition is false."

In Ruby, the conditional expression can be a simple comparison, as illustrated above with the **==** (double equals) operator. Or **if** can be followed by a variable that has been assigned a boolean value. Or you can call a method that returns a boolean result.

Ternary Operator

A basic conditional structure might look like this:

```ruby
if date.nil?
  @date = Date.today
else
  @date = date
end
```

We test if **date** is undefined (nil). If nil, we assign today's date to the instance variable **@date**. If **date** is already assigned a value, we assign it to the instance variable **@date**. This is useful in the **initialize(name,date)** method in our example code because we want to set today's date as the default value for the instance variable **@date** if the parameter **date** is nil.

Ruby developers like to keep their code tight and compact. So you'll see a condensed version of this conditional structure often, particularly when a default value must be assigned.

This compact conditional syntax is named the *ternary operator* because it has three components. Here is the syntax:

```ruby
condition ? value_if_true : value_if_false
```

Here is the ternary operator we use in our example code:

```ruby
@date = date.nil? ? Date.today : date
```

This is another example of Ruby syntax that you must learn to recognize by sight because it is difficult to interpret if you have never seen it before.

For more Ruby code that has been condensed into obscurity, see an article on Ruby Golf. Ruby golf is the sport of writing code that uses as few characters as possible.

Interpolation

Rubyists love to find special uses for orthography such as hashmarks and curly braces. It seems Rubyists feel sorry for punctuation marks that don't get much use in the English language and like to give them new jobs.

We already know that we can assign a string to a variable:

```ruby
name = 'Foobar Kadigan'
```

We can also perform "string addition" to concatenate strings. Here we add an honorific, a space, and a name:

```ruby
@honorific = 'Mr.'
@name = 'Foobar Kadigan'
titled_name = @honorific + ' ' + @name
=> "Mr. Foobar Kadigan"
```

Single quote marks indicate a string. In the example above, we enclose a space character within quote marks so we add a space to our string.

You can eliminate the ungainly mix of plus signs, single quote marks, and space characters in the example above.

Use double quote marks and you can perform *interpolation*, which gives a new job to the hashmark and curly brace characters:

```ruby
@honorific = 'Mr.'
@name = 'Foobar Kadigan'
titled_name = "#{@honorific} #{@name}"
=> "Mr. Foobar Kadigan"
```

The hashmark indicates any expression within the curly braces is to be evaluated and returned as a string. This only works when you surround the expression with double quote marks.

Interpolation is cryptic when you first encounter the syntax, but it streamlines string concatenation.

Access Control

Any method you define will return a result.

Sometimes you want to create a method that only can be used by other methods in the same class. This is common when you need a simple utility method that is used by several other methods.

Any methods that follow the keyword **private** should only be used by methods in the same class (or a subclass).

```
private
```

You often see private methods in Rails. Ruby provides a *protected* keyword as well, but the difference between *protected* and *private* is subtle and *protected* is seldom seen in Rails applications.

Hash

Our example code includes a private method named **famous_birthdays** that returns a collection of names and birthdays of famous musicians.

Computers have always been calculation machines; they are just as important in managing collections.

One important type of collection is named a Hash. A Hash is a data structure that associates a key to some value. You retrieve the value based upon its key. This construct is called a *dictionary*, an *associative array*, or a *map* in other languages. You use the key to "look up" a value, as you would look up a definition for a word in a dictionary.

You'll recognize a Hash when you see curly braces (again, Rubyists give a job to under-utilized punctuation marks).

```
birthdays = {
  'Ludwig van Beethoven' => Date.new(1770,12,16),
  'Dave Brubeck' => Date.new(1920,12,6),
  'Buddy Holly' => Date.new(1936,9,7),
  'Keith Richards' => Date.new(1943,12,18)
}
```

Rubyists also like to create novel uses for mathematical symbols. The combination of an **=** (equals) sign and **>** (greater than) sign is called a *hashrocket*. The **=>** (hashrocket) operator associates a key and value pair in a Hash.

Ruby 1.9 introduced a new way to associate key and value pairs in a Hash:

```
birthdays = {
  beethoven: Date.new(1770,12,16),
  brubeck: Date.new(1920,12,6),
  holly: Date.new(1936,9,7),
  richards: Date.new(1943,12,18)
}
```

Here, instead of using a string as the key, we are using Ruby symbols, which enable faster processing. The **:** (colon) character associates the key and value.

Ordinarily, a symbol is defined with a leading colon character. In a Hash, a trailing colon makes a string into a symbol.

If you want to transform a string containing spaces into a symbol in a Hash, you can do it, though the syntax is awkward:

```
birthdays = {
  :'Ludwig van Beethoven' => Date.new(1770,12,16)
}
```

Whether with colons or hashrockets, you'll often see Hashes used in Rails.

Array

An *Array* is a list. Arrays can hold objects of any data type. In fact, arrays can contain a mix of different objects. For example, an array can contain a string and another array (this is an example of a *nested array*).

An array can be instantiated with square brackets:

```ruby
born_in_december = [ ]
```

We can populate the array with values when we create it:

```ruby
my_list = ['apples', 'oranges']
```

If we don't want to use quote marks and commas to separate strings in a list, we can use the **%w** syntax:

```ruby
my_list = %w( apples oranges )
```

We can add new elements to an array with a **push** method:

```ruby
my_list = Array.new
=> []
my_list.push 'apples'
=> ["apples"]
my_list.push 'oranges'
=> ["apples", "oranges"]
```

In our example code, we use the **<<** *shovel operator* to add items to the array:

```
born_in_december << name
```

A Ruby array has close to a hundred available methods, including operations such as **size** and **sort**. See the Ruby API for a full list.

Iterator

Of all the methods available for a Ruby collection such as Hash or Array, the *iterator* may be the most useful.

You'll recognize an iterator when you see the **each** method applied to a Hash or Array:

```
famous_birthdays.each
```

The **each** keyword is always followed by a block of code. Each item in an Array, or key-value pair in a Hash, is passed to the block of code to be processed.

Block

You can recognize a *block* in Ruby when you see a **do ... end** structure. A block is a common way to process each item when an iterator such as **each** is applied to a Hash or Array.

In our example, we iterate over the **famous_birthdays** hash:

```
famous_birthdays.each do |name, date|
  .
  .
  .
end
```

Within the two pipes (or bars), we assign the key and value to two variables.

The block is like an unnamed method. The two variables are available only within the block. As each key-value pair is presented by the iterator, the variables are assigned, and the statements in the block are executed.

In our example code, we evaluate each date in the **famous_birthdays** hash to determine if the musician was born in December. When we find a December birthday, we add the name of the musician to the **born_in_december** array:

```
famous_birthdays.each do |name, date|
  if date.month == 12
    born_in_december << name
  end
end
```

When you use a block within a method, any variable in your method is available within the block. That's why we can add **name** to the array **born_in_december**.

Computer scientists consider a block to be a programming language construct called a *closure*. Ruby has other closures, including the *proc* (short for procedure) and the *lambda*. Though blocks are common you'll seldom see procs or lambdas in ordinary Rails code. They are more common in the Rails source code where advanced programming techniques are used more frequently.

The key point to know about a block (or a proc or a lambda) is that it works like a method. Though you don't see a method definition, you can use a block to evaluate a sequence of statements and obtain a result.

Rails and More Keywords

We've looked at only a few of the keywords and constructs you will see in Ruby code. The exercise has improved your Ruby literacy, so you'll have an easier time reading Ruby code.

Nothing in the exercise is Rails. The example code only uses keywords from the Ruby API.

Rails has its own API, with hundreds of classes and methods. The Rails API uses the syntax and keywords of the Ruby language to construct new classes and create new keywords that are specific to Rails and useful for building web applications.

We say Ruby is a general-purpose language because it can be used for anything. Rails is a *domain-specific language* (DSL) because it is used only by people building web applications (in this sense, "domain" means area or field of activity). Ruby is a great language to use for building a DSL, which is why it was used for Rails. Unlike some other programming languages, Ruby easily can be extended or tweaked. For example, developers can redefine classes, add extra methods to existing classes, and use the special **method_missing** method to handle method calls that aren't previously defined. Software architects call this *metaprogramming* which simply means clever programming that twists and reworks the programming language.

When you add a gem to a Rails project, you'll add additional keywords. Some of the most powerful gems add their own DSLs to your project. For example, the Cucumber gem provides a DSL for turning user stories into automated tests.

Adding Rails, additional gems, and DSLs provides powerful functionality at the cost of complexity. But it all conforms to the syntax of the Ruby language. As you learn to recognize Ruby keywords and language structures, you'll be able to pick apart the complexity and make sense of any code.

More Ruby

To develop your proficiency as a Rails developer, I hope you will make an effort to learn Ruby as you learn Rails. Don't be lazy; when you encounter a bit of Ruby you don't understand, make an effort to find out what is going on. Spend time with a Ruby textbook or interactive course when you work on Rails projects.

Collaborative Learning

The best way to learn Ruby is to actually use it. That's the concept behind this site:

- Exercism.io

With Exercism, you'll work though code exercises and get feedback from other learners.

Online Tutorials

- TryRuby.org - free browser-based interactive tutorial from Code School

- Codecademy Ruby Track - free browser-based interactive tutorials from Codecademy

- Ruby Koans - free browser-based interactive exercises from Jim Weirich and Joe O'Brien

- Ruby in 100 Minutes - free tutorial from JumpstartLab

- Code Like This - free tutorials by Alex Chaffee

- RailsBridge Ruby - basic introduction to Ruby

- CodeSchool Ruby Track - instructional videos with in-browser coding exercises

Books

- Learn To Program - free ebook by Chris Pine

- Learn To Program - expanded $18.50 ebook by Chris Pine

- [Learn Code the Hard Way](#) - free from Zed Shaw and Rob Sobers

- [Beginning Ruby](#) - by Peter Cooper

- [Programming Ruby](#) - by Dave Thomas, Andy Hunt, and Chad Fowler

- [Eloquent Ruby](#) - by Russ Olsen

- [Books by Avdi Grimm](#), including *Confident Ruby* and *Objects on Rails*.

Newsletters

- [Practicing Ruby](#) - $8/month for access to over 90 helpful articles on Ruby

- [RubySteps](#) - weekly lessons by email from Pat Maddox

Screencasts

- [RubyTapas](#) - $9/month for access to over 100 screencasts on Ruby

Git

There's no need to save the file **lib/example.rb** file we created to learn Ruby. You can simply delete the file:

```
$ rm lib/example.rb
```

Check the Git status to make sure the file is gone:

```
$ git status
# On branch master
nothing to commit, working directory clean
```

We've cleaned up after our Ruby exercise.

From here on, we're done with silly code examples. No more fooling around. With the next chapter, we start building a real-world Rails website.

Chapter 16

Layout and Views

In previous chapters we created a dynamic home page and learned techniques for troubleshooting.

In this chapter we'll look closely at view files, particularly the application layout, so we can organize the design of our web pages. We'll also learn how to add a CSS stylesheet to improve the graphic design of our web pages.

This chapter covers a lot of ground, so take a break before jumping in, or pace yourself to absorb it all.

Template Languages

HTML is intended for markup, which means applying formatting to a text file. For a web application, ordinary HTML is not sufficient; we need to mix in Ruby code. We'll use a *templating language* that gives us a syntax for mixing HTML tags and Ruby code. The Ruby code will be processed by a *templating engine* built into Rails. The output will be pure HTML sent to the browser.

The most popular templating language available for Rails is *ERB*, Embedded Ruby, which is the Rails default.

In the "Concepts" chapter in Book One, you learned that components of Rails can be mixed for different "stacks." Some developers substitute Haml or Slim for ERB. We'll use ERB in this book because it is the most popular.

Introducing the Application Layout

We've already created the view file for our home page.

The file **app/views/visitors/new.html.erb** looks like this:

```
<h3>Home</h3>
<p>Welcome to the home of <%= @owner.name %>.</p>
<p>I was born on <%= @owner.birthdate %>.</p>
<p>Only <%= @owner.countdown %> days until my birthday!</p>
```

The first line in the file contains an HTML heading tag, **<h3>**, with headline text, "Home."

When you used the browser diagnostic view to see the HTML file received by the server, you saw this:

```
<!DOCTYPE html>
<html>
<head>
<title>LearnRails</title>
<meta name="csrf-param" content="authenticity_token" />
<meta name="csrf-token" content="..." />
<link rel="stylesheet" media="all" href="/assets/application.css?body=1" data-turbolinks-track=
<script src="/assets/jquery.js?body=1" data-turbolinks-track="reload"></script>
<script src="/assets/jquery_ujs.js?body=1" data-turbolinks-track="reload"></script>
<script src="/assets/turbolinks.js?body=1" data-turbolinks-track="reload"></script>
<script src="/assets/action_cable.js?body=1" data-turbolinks-track="reload"></script>
<script src="/assets/cable.js?body=1" data-turbolinks-track="reload"></script>
<script src="/assets/application.js?body=1" data-turbolinks-track="reload"></script>
</head>
<body>

<h3>Home</h3>
<p>Welcome to the home of Foobar Kadigan.</p>
<p>I was born on 1990-09-22.</p>
```

```
<p>Only 126 days until my birthday!</p>


</body>
</html>
```

If you've built websites before, you'll recognize the HTML file conforms to the HTML5 specification, with a **DOCTYPE**, **<head>** and **<body>** tags, and miscellaneous tags in the HEAD section, including a title and various CSS and JavaScript assets.

If you look closely, you'll see some HTML attributes you might not recognize, for example the **data-turbolinks-track** attribute. That is added by Rails to support turbolinks, for faster loading of webpages.

For the most part, everything is ordinary HTML. But only part of it originates from the view file we've created for our home page.

Where did all the extra HTML come from?

The final HTML file is more than twice the size of the view file.

The additional tags come from the default *application layout* file.

Rails has combined the Visitors#New view with the default application layout file. To be precise, the **new** method in the Visitors controller has rendered the home page by combining two files:

- the application layout file **app/views/layouts/application.html.erb**

- the Visitors#New template in the file **app/views/visitors/new.html.erb**

Let's examine the application layout file.

Open the file **app/views/layouts/application.html.erb**:

```
<!DOCTYPE html>
<html>
  <head>
    <title>LearnRails</title>
    <%= csrf_meta_tags %>

    <%= stylesheet_link_tag 'application', media: 'all',
        'data-turbolinks-track': 'reload' %>
    <%= javascript_include_tag 'application',
        'data-turbolinks-track': 'reload' %>
  </head>

  <body>
    <%= yield %>
  </body>
</html>
```

Static pages delivered from the **public** folder do not use the default application layout. But every page generated by the model-view-controller architecture in the **app/** folder incorporates the default application layout, unless you specify otherwise.

The default application layout is where you put HTML that you want to include on every page of your website.

Remember when we looked at the hidden code in the controller that renders a view? The controller uses the **render** method to combine the view file with the application layout.

Here's the Visitors controller, again, with the hidden **render** method revealed:

```
class VisitorsController < ApplicationController

  def new
    @owner = Owner.new
    render 'visitors/new'
  end

end
```

The **render** method combines the **app/views/visitors/new.html.erb** view file with the **app/views/layouts/application.html.erb** application layout.

Alternatively, you could tell the controller to render the view without any application layout:

```
render 'visitors/new', :layout => false
```

Or you could specify an alternative layout file, for example **app/views/layouts/special.html.**

```
render 'visitors/new', :layout => 'special'
```

An alternative layout can be useful for special categories of pages, such as administrative pages or landing pages.

We won't use alternative layouts in this tutorial application, but it's good to know they are an option. The reference RailsGuides: Layouts and Rendering in Rails explains more about using alternative layouts.

Yield

How does the **render** method insert the view file in the application layout?

Notice that the default application layout contains the Ruby keyword **yield**.

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

The **yield** keyword is replaced with a view file that is specific to the controller and action, in this case, the **app/views/visitors/new.html.erb** view file.

The content from the view is inserted where you place the **yield** keyword.

Yield Variations

We won't do it, but you could also use the `yield` keyword to insert a sidebar
or a footer.

Rails provides ways to insert content into a layout file at different places. The
`content_for` method is helpful when your layout contains distinct regions
such as sidebars and footers that should contain their own blocks of content.

For example, you could create an application layout that includes a sidebar.
This is just an example, so don't add it to the application you are building:

```erb
<!DOCTYPE html>
<html>
<head>
  <title>LearnRails</title>
  <%= csrf_meta_tags %>
  <%= stylesheet_link_tag 'application', media: 'all',
      'data-turbolinks-track': 'reload' %>
  <%= javascript_include_tag 'application',
      'data-turbolinks-track': 'reload' %>
</head>
<body>
  <div class="main">
    <%= yield %>
  </div>
  <div class="sidebar">
    <%= yield :sidebar %>
  </div>
</body>
</html>
```

This view file provides both the main content and a sidebar:

```erb
<% content_for :sidebar do %>
  <h3>Contact Info</h3>
  <p>Email: me@example.com</p>
<% end %>
<h3>Main</h3>
<p>Welcome!</p>
```

This section gets inserted at the `<%= yield :sidebar %>` location:

```
<% content_for :sidebar do %>
  <h3>Contact Info</h3>
  <p>Email: me@example.com</p>
<% end %>
.
.
.
```

The rest of the file gets inserted at the main `<%= yield %>` location.

Again, don't add this to your application. I'm just offering it as an example of multiple `yield` statements.

The reference RailsGuides: Layouts and Rendering in Rails explains more about using `yield` and `content_for`.

ERB Delimiters

Earlier, we saw ERB `<%= ...  %>` delimiters allow us to insert Ruby expressions which are replaced by the result of evaluating the code. Here is an example that displays the number 4

```
<%= 2 + 2 %>
```

Look closely and you'll see this ERB delimiter is slightly different:

```
<% 3.times do %>
  <li>list item</li>
<% end %>
```

An ERB delimiter that does not contain the `=` (equals) sign will execute Ruby code but will not display the result. It is commonly used to add Ruby blocks to HTML code, so you'll often see `do` and `end` statements within ERB `<% ... %>` delimiters. The example above will create three list items, like this:

```
<li>list item</li>
<li>list item</li>
<li>list item</li>
```

A third version of the ERB delimiter syntax is rarely seen:

```
<%# this is a comment %>
```

It is only used for adding comments. The expression within the ERB `<%# ...
%>` delimiters will not execute and will not appear when the page is output as
HTML.

Introducing View Helpers

We can use ERB delimiters to create Rails *view helpers*.

We've seen how ERB delimiters can enclose Ruby code.

In the application layout file, the `<%= ...   %>` delimiters don't include any-
thing that looks like Ruby code. For example, we see `<%= csrf_meta_tags
%>` which seems to be neither HTML nor anything from the Ruby API. In fact,
this expression is Ruby code, but it is from the Rails API and only found in
Rails applications.

Ruby is an ideal choice for a web application development platform such as
Rails because it can easily be used to create a *domain-specific language* (or
DSL). Much of Rails is a domain-specific language. The Smalltalk program-
ming language was famous for its slogan "Code should read like a conversa-
tion." Ruby, which borrows much from Smalltalk, makes it easy to add new
words to the conversation. We can add new keywords that produce complex
behaviour, creating entire new APIs such as Rails. Ruby makes it easy for the
Rails core team to add keywords such as `csrf_meta_tags` that are additions
to the Ruby language.

In this case, Ruby's ability to produce a domain-specific language gives us Rails *view helpers*.

Think of Rails view helpers as "macros to generate HTML." You may have used macros to automate a series of commands in World of Warcraft or other games. If you're an office worker, you may have used macros in Microsoft Word or Excel. A Rails view helper is a keyword that expands into a longer string of HTML tags and content.

In this case, the **csrf_meta_tags** view helper expands into two lines of HTML:

```
<meta name="csrf-param" content="authenticity_token" />
<meta name="csrf-token" content="bhd8ZZJ7dsMza8ZT1Csm5IuCIxkNNq==" />
```

Why do we need this cryptic code? It turns out that almost any website that accepts user input via a form is vulnerable to a security bug (an *exploit*) named a cross-site request forgery. To prevent rampant CSRF exploits, the Rails core team includes the **csrf_meta_tags** view helper in the default application layout. Rails provides a number of similar features that make websites more secure.

A Rails view file becomes much less mysterious when you realize that many of the keywords you see are view helpers. Strange new keywords may be part of the Rails API. Or they may be provided by gems you've added (gem developers often use the Ruby DSL capability to create new keywords). Think of it this way: Ruby gives developers the power to create an unlimited number of new "HTML tags." These tags are not really HTML because they are not part of the HTML specification. But they serve as shortcuts to produce complex snippets of HTML and content.

Now that we've learned about view helpers, we can start building our default application layout.

The Rails Layout Gem

Every Rails application needs a well-designed application layout. The Rails default starter application, which we get when we run **rails new**, provides a barebones application layout. It is purposefully simple so developers can add the code they need to accommodate any front-end framework (we'll look closely at front-end frameworks in the next chapter).

In this chapter we'll start with a simple application layout file, adding a little CSS for simple styling. In the next chapter, we'll upgrade the application layout file to use the Bootstrap front-end framework.

To make it easy, we'll use the rails_layout gem to generate files for an application layout. In this chapter, we'll use the rails_layout gem to create our basic layout and CSS files. In the next chapter, we'll use the rails_layout gem to create layout files for Bootstrap.

In your **Gemfile**, you've already added:

```
gem 'rails_layout'
```

and previously run **$ bundle install**.

Rails provides the **rails generate** command to run simple scripts that are packaged into gems.

The rails_layout gem uses the **rails generate** command to set up files we need. Run:

```
$ rails generate layout:install simple --force
```

The **--force** argument will force the gem to replace the existing **app/views/layouts/applicat** file.

If you have the **app/views/layouts/application.html.erb** file open in your text editor, it will change.

The rails_layout gem will rename the file:

- **app/assets/stylesheets/application.css**

to:

- **app/assets/stylesheets/application.css.scss**

The gem will add (or modify) five files:

- **app/views/layouts/application.html.erb**

- **app/assets/stylesheets/simple.css**

- **app/views/layouts/_messages.html.erb**

- **app/views/layouts/_navigation.html.erb**

- **app/views/layouts/_navigation_links.html.erb**

Examining these files closely will reveal a great deal about the power of Rails. We'll dedicate the rest of this chapter to exploring the contents of these files.

Basic Boilerplate

Open the file **app/views/layouts/application.html.erb**:

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title><%= content_for?(:title) ? yield(:title) : "Learn Rails" %></title>
    <meta name="description" content=
```

```
      "<%= content_for?(:description) ? yield(:description) : "Learn Rails" %>">
    <%= stylesheet_link_tag 'application', media: 'all',
      'data-turbolinks-track' => 'reload' %>
    <%= javascript_include_tag 'application', 'data-turbolinks-track' => 'reload' %>
    <%= csrf_meta_tags %>
  </head>
  <body>
    <header>
      <%= render 'layouts/navigation' %>
    </header>
    <main role="main">
      <%= render 'layouts/messages' %>
      <%= yield %>
    </main>
  </body>
</html>
```

Some of this code is already familiar.

You'll recognize the standard HTML **DOCTYPE**, **<head>**, and **<body>** tags.

We've already discussed the **yield** keyword.

We've seen the **<%= ... %>** delimiters surrounding the **csrf_meta_tags** view helper:

- **csrf_meta_tags** - generates **<meta>** tags that prevent cross-site request forgery

The rest of the file may be unfamiliar. We'll examine it line by line.

Adding Boilerplate

Webmasters who build static websites are accustomed to setting up web pages with "boilerplate," or basic templates for a standard web page. The well-known HTML5 Boilerplate project has been recommending "best practice" tweaks to web pages since 2010. Very few of the HTML5 Boilerplate recommendations are relevant for Rails developers, as Rails already provides almost everything

required. We'll discuss one important boilerplate item and a few "nice to have" extras.

If you want to learn more, the article HTML5 Boilerplate for Rails Developers looks at the recommendations.

Viewport

The `viewport` metatag improves the presentation of web pages on mobile devices. Setting a viewport tells the browser how content should fit on the device's screen. The tag is required for either Bootstrap or Zurb Foundation front-end frameworks.

The `viewport` metatag looks like this:

```
<meta name="viewport" content="width=device-width, initial-scale=1.0">
```

Apple's developer documentation on Configuring the Viewport provides details.

Title and Description

If you want to maximize traffic to your website, you should make your web pages search-engine friendly. That means adding *title* and *description* metatags. Google uses contents of the title tag to display titles in search results. And it will sometimes use the content of a description metatag in search results snippets. See Google's explanation of how it uses Site Title and Description. Good titles and descriptions improve clickthrough from Google searches.

Title and description looks like this:

```
<title><%= content_for?(:title) ? yield(:title) : "Learn Rails" %></title>
<meta name="description" content="<%= content_for?(:description) ?
    yield(:description) : "Learn Rails" %>">
```

The rails_layout gem has created a default title and description based on our project name.

Later in the tutorial, we'll see how to use a **content_for** statement to set a title and description for each individual page.

The code is complex if you haven't seen advanced Ruby before. It uses the Ruby ternary operator which maximizes compactness at the price of introducing obscurity. You'll recall from the "Just Enough Ruby" chapter that it is a fancy conditional statement that says, "if **content_for?(:title)** is present in the view file, use **yield(:title)** to include it, otherwise just display 'Learn Rails'."

Asset Pipeline

You may have noticed these Rails helper methods:

- **stylesheet_link_tag**

- **javascript_include_tag**

These are tags that add CSS and JavaScript to the web page using the Rails *asset pipeline*.

The Rails *asset pipeline* utility is one of the most powerful features of the platform. It offers convenience to the developer and helps organize an application; more importantly, it improves the speed and responsiveness of any complex website. If you're going to do any front-end development with CSS or JavaScript in Rails, you must understand the Rails asset pipeline. Here's how it works.

Assets Without Rails

When building non-Rails websites, webmasters add JavaScript to a page using the **`<script>`** tag. For every JavaScript file, they add an additional **`<script>`** tag, so a page HEAD section looks like this:

```html
<!DOCTYPE html>
<html>
<head>
  <title>Page that uses multiple JavaScript files</title>
  <script src="jquery.js" type="text/javascript"></script>
  <script src="jquery.plugin.js" type="text/javascript"></script>
  <script src="custom.js" type="text/javascript"></script>
</head>
```

The same is true for CSS files in non-Rails websites. You add a **`<link>`** tag for each stylesheet file. With multiple stylesheets, the HEAD section of your application layout might look like this:

```html
<!DOCTYPE html>
<html>
<head>
  <title>Page that uses multiple CSS files</title>
  <link href="core.css" rel="stylesheet" type="text/css" />
  <link href="site.css" rel="stylesheet" type="text/css" />
  <link href="custom.css" rel="stylesheet" type="text/css" />
</head>
```

If you want to handle CSS and JavaScript without Rails, you can place your files in the **public** folder. If you do so, every time you add a JavaScript or CSS file, you must modify the application layout file. Instead, use the asset pipeline and simplify this.

Assets With Rails

The asset pipeline consists of two folders:

- **app/assets/javascripts/**

- **app/assets/stylesheets/**

Any JavaScript and CSS file you add to these folders is automatically added to every page.

In development, when the web browser makes a page request, the files in the asset pipeline folders are combined together and concatenated as single large files, one for JavaScript and one for CSS.

If you examine the application layout file, you'll see the tags that perform this service:

```
<%= stylesheet_link_tag 'application', media: 'all',
    'data-turbolinks-track' => 'reload' %>
<%= javascript_include_tag 'application', 'data-turbolinks-track' => 'reload' %>
```

The HTML delivered to the browser looks like this:

```
<link href="/assets/application.css" media="all" rel="stylesheet" type="text/css" />
<script src="/assets/application.js" type="text/javascript"></script>
```

Using the asset pipeline, there is no need to modify the application layout file each time you create a new JavaScript or CSS file. Create as many files as you need to organize your JavaScript or CSS code and, in production, you'll automatically get one single file delivered to the browser. In development mode, Rails continues to deliver multiple files for easier debugging.

In production, there's a big performance advantage with the asset pipeline. Requesting files from the server is a time-consuming operation for a web browser, so every extra file request slows down the browser. The Rails asset pipeline eliminates the performance penalty of multiple **<script>** or **<link>** tags. The Rails asset pipeline also compresses JavaScript and CSS files for faster page loads.

The asset pipeline is an example of a Rails convention that helps developers build complex websites. It is not needed for a simple website that uses a few JavaScript or CSS files. But it is beneficial on bigger projects.

The asset pipeline has limitations, especially for projects with complex front-end JavaScript. Rails developers who integrate JavaScript front-end frameworks such as React or Angular often choose not to use the asset pipeline. Instead they rely on JavaScript tooling such as Webpack to manage complex JavaScript assets.

Now that you understand the purpose of the Rails asset pipeline, let's look at more of the code in the default application layout file.

Navigation Links

Every website needs navigation links.

You can add navigation links directly to your application layout but many Rails developers prefer to create a partial template —-a "partial"—-to better organize the default application layout.

Introducing Partials

A *partial* is similar to any view file, except the filename begins with an underscore character. Place the file in any view folder and you can use the **render** keyword to insert the partial.

We're not going to add a footer to our tutorial application, but here is how we could do it. We'd use the **render** keyword with a file named **app/views/layouts/_footer.htm**

```
<%= render 'layouts/footer' %>
```

Notice that you specify the folder within the **app/views/** directory with a trun-

cated version of the filename. The `render` method doesn't want the _ underscore character or the `.html.erb` file extension. That can be confusing; it makes sense when you remember that Rails likes "convention over configuration" and economizes on extra characters when possible.

We're not going to add a footer to our application, but we will add navigation links by using a partial. First, let's learn about *link helpers*.

Introducing Link Helpers

There's no rule against using raw HTML in our view files, so we could create a partial for navigation links that uses the HTML `<a>` anchor tag like this:

```
<ul class="nav">
  <li><a href="/">Home</a></li>
  <li><a href="/about">About</a></li>
  <li><a href="/contact">Contact</a></li>
</ul>
```

Rails gives us another option, however. We can use the Rails `link_to` view helper instead of the HTML `<a>` anchor tag. The Rails `link_to` helper eliminates the crufty `<>` angle brackets and the unnecessary `href=""`. More importantly, it adds a layer of abstraction, using the routing configuration file to form links. This is advantageous if we make changes to the location of the link destinations. Earlier, when we created a static "About" page, we first set the **config/routes.rb** file with a route to the "About" page: `root to: redirect('/about.html')`. Later we removed the static "About" page and set the **config/routes.rb** file with a route to the dynamic home page: `root to: 'visitors#new'`. If we used the raw HTML `<a>` anchor tag, we'd have to change the raw HTML everywhere we had a link to the home page. Using the Rails `link_to` helper, we name a route and make any changes once, in the **config/routes.rb** file.

When you use the Rails `link_to` helper, you'll avoid the problem of link maintenance that webmasters face on static websites. Some webmasters like to use

absolute URLs, specifying a host name in the link, for example **http://www.example.co**
Absolute URLs are a headache when moving the site, for example from **staging.example**
to **www.example.com**. The problem is avoided by using *relative* URLs, such
as **/about.html**, **about.html**, or even **../about.html**. But relative URLs
are fragile, and moving files or directories often results in overlooked and bro-
ken links. Instead, with the Rails **link_to** helper, you always get the destina-
tion location specified in the **config/routes.rb** file.

Navigation Partial

Examine the **app/views/layouts/application.html.erb** and you'll see the use
of the navigation partial.

We include the navigation partial in our application layout with the expression:

```
.
.
.
<%= render 'layouts/navigation' %>
.
.
.
```

Open the file **app/views/layouts/_navigation.html.erb**:

```
<ul class="nav">
  <li><%= link_to 'Home', root_path %></li>
  <%= render 'layouts/navigation_links' %>
</ul>
```

You'll see the **link_to** helper.

Here the **link_to** helper takes two parameters. The first parameter is the string
displayed as the anchor text (**'Home'**). The second parameter is the route. In
this case, the route **root_path** has been set in the **config/routes.rb** file.

The navigation partial includes another partial, which we'll call the navigation links partial:

```
.
.
.
  <%= render 'layouts/navigation_links' %>
.
.
.
```

This demonstrates that one partial can include another partial, so that partials can be "nested."

Navigation Links Partial

In our simple application, there's no obvious reason to nest another partial. But we'll see in the next chapter that it is convenient, because we can isolate the complex markup required by Bootstrap from the simple list of links we need for navigation.

Open the file **app/views/layouts/_navigation_links.html.erb**:

```
<%# add navigation links to this file %>
```

As we add pages to our application, we'll add links to this file.

For now, we have nothing to add.

Flash Messages

Rails provides a standard convention to display alerts (including error messages) and other notices (including success messages), called a *flash message*.

The name comes from the term "flash memory" and should not be confused with the "Adobe Flash" web development platform that was once popular for animated websites. The flash message is documented in the RailsGuides: Action Controller Overview.

Here's a flash message you might see after logging in to an application:

Signed in successfully.   ✕

Figure 16.1: Flash message in Rails.

It is called a "flash message" because it appears on a page temporarily. When the page is reloaded or another page is visited, the message disappears.

Typically, you will see only one flash message on a page. But there is no limit to the number of flash messages that can appear on a page.

Creating Flash Messages

Flash messages are created in a controller. For example, we can add messages to the home page by modifying the file **app/controllers/visitors_controller.rb** like this:

```ruby
class VisitorsController < ApplicationController

  def new
    @owner = Owner.new
    flash[:notice] = 'Welcome!'
    flash[:alert] = 'My birthday is soon.'
  end

end
```

If you test the application after adding the messages to the VisitorsController, you'll see two flash messages appear on the page.

Rails provides the **flash** object so that messages can be created in the controller and displayed on the rendered web page.

In this example, we create a flash message by associating the object **flash[:notice]** with the string **'Welcome!'**. We can assign other messages, such as **flash[:alert]** or even **flash[:warning]**. In practice, Rails uses only **:notice** and **:alert** as flash message keys so it is wise to stick with just these.

Flash and Flash Now

You can control the persistence of the flash message by choosing from two variants of the **flash** directive.

Use **flash.now** in the controller when you immediately render a page, for example with a **render :new** directive. With **flash.now**, the message will vanish after the user clicks any links.

Use the simple variant, **flash**, in the controller when you redirect to another page, for example with a **redirect_to root_path** directive. If you use **flash.now** before a redirect, the user will not see the flash message because **flash.now** does not persist through redirects or links. If you use the simple **flash** directive before a **render** directive, the message will appear on the rendered page and reappear on a subsequent page after the user clicks a link.

In our example above, we really need to use the **flash.now** variant because the controller provides a hidden **render** method. Update the file **app/controllers/visitors_cont**

```
class VisitorsController < ApplicationController

  def new
    @owner = Owner.new
    flash.now[:notice] = 'Welcome!'
    flash.now[:alert] = 'My birthday is soon.'
  end

end
```

Using **flash.now** will make sure the message only appears on the rendered

page and will not persist after a user follows a link to a new page.

If you ever see a "sticky" flash message that won't go away, you need to use `flash.now` instead of `flash`.

Explaining the Ruby Code

If you're new to programming in Ruby, it may be helpful to learn how the `flash` object works.

The `flash` object is a Ruby *hash*.

You'll recall from the "Just Enough Ruby" chapter that a hash is a data structure that associates a key to some value. You retrieve the value based upon its key. This construct is called a *dictionary* in other languages, which is appropriate because you use the key to "look up" a value, as you would look up a definition for a word in a dictionary.

Hash is a type of *collection*. Presumably, the Rails core contributors who implemented the code chose to use a collection so that a page could be given multiple flash messages. Because we have a collection with (possibly) multiple messages, we need to retrieve each message one at a time.

We learned earlier that all collections support an *iterator* method named `each`. Iterators return all the elements of a collection, one after the other. The iterator returns each key-value pair, item by item, to a *block*. In Ruby, a block is delimited by `do` and `end` or `{ }` braces. You can add any code to a block to process each item from the collection.

Here is simple Ruby code to iterate through a `flash` object, outputting each flash message in an HTML `div` tag and applying a CSS class for styling:

```ruby
flash.each do |key, value|
  puts '' + value + ''
end
```

In this simple example, we use `each` to iterate through the flash hash, retrieving

a **key** and **value** that are passed to a block to be output as a string. We've chosen the variable names **key** and **value** but the names are arbitrary. In the next example, we'll use **name** and **msg** as variables for the key-value pair. The output string will appear as HTML like this:

```
<div class="notice">Welcome!</div>
<div class="alert">My birthday is soon.</div>
```

Let's continue examining our layout files.

The Flash Messages Partial

Flash messages are a very useful feature for a dynamic website.

Code to display flash messages can go directly in your application layout file or you can use a partial.

Examine the file **app/views/layouts/_messages.html.erb**:

```
<% flash.each do |name, msg| %>
  <% if msg.is_a?(String) %>
    <%= content_tag :div, msg, :class => "flash_#{name}" %>
  <% end %>
<% end %>
```

It improves on our simple Ruby example in several ways. First, the expression **if msg.is_a?(String)** serves as a test to make sure we only display messages that are strings. Second, we use the Rails **content_tag** view helper to create the HTML **div**. The **content_tag** helper eliminates the messy soup of angle brackets and quote marks we used to create the HTML output in the example above. Finally, we apply a CSS **class** and combine the word "flash" with "notice" or "alert" to make the CSS class.

We include the flash messages partial in our application layout with the expression:

```
.
.
.
<%= render 'layouts/messages' %>
.
.
.
```

HTML5 Elements

Let's look again at the **app/views/layouts/application.html.erb** file.

To complete our examination of the application layout file, we'll look at a few structural elements. These elements are not unique to a Rails application and will be familiar to anyone who has done front-end development.

Notice the tags that are structural elements in the HTML5 specification:

- **<header>**

- **<main>**

These elements add structure to a web page. The tags don't add any new behavior but make it easier to determine the structure of the page and apply CSS styles.

We wrap the navigation partial in the **<header>** tag:

```
<header>
  <%= render 'layouts/navigation' %>
</header>
```

The **<header>** tag is typically used for branding or navigation.

Notice the *main tag*:

```
<main role="main">
  <%= render 'layouts/messages' %>
  <%= yield %>
</main>
```

We wrap our messages partial and `yield` expression in a `<main role="main">` element. The `<main>` tag is among the newest HTML5 elements (see the W3C specification for details). From the specification: "The main content area of a document includes content that is unique to that document and excludes content that is repeated across a set of documents such as site navigation links, copyright information, site logos." We follow the advice of the specification and wrap our unique content in the `<main>` tag.

The specification recommends, "Authors are advised to use ARIA role='main' attribute on the main element until user agents implement the required role mapping." ARIA, the Accessible Rich Internet Applications Suite, is a specification to make web applications more accessible to people with disabilities. That means the `role="main"` attribute is there for any web browsers that don't yet recognize the `<main>` tag, and may help people with disabilities.

We could add a `<footer>` tag. It typically contains links to copyright information, legal disclaimers, or contact information. We don't have a footer in our tutorial application but you can add the `<footer>` tag, with additional content, if you want.

Application Layout

Our application layout is complete. We don't have to add anything because the rails_layout gem has created everything we need.

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
```

```erb
    <title><%= content_for?(:title) ? yield(:title) : "Learn Rails" %></title>
    <meta name="description" content=
      "<%= content_for?(:description) ? yield(:description) : "Learn Rails" %>">
    <%= stylesheet_link_tag 'application', media: 'all',
      'data-turbolinks-track' => 'reload' %>
    <%= javascript_include_tag 'application', 'data-turbolinks-track' => 'reload' %>
    <%= csrf_meta_tags %>
  </head>
  <body>
    <header>
      <%= render 'layouts/navigation' %>
    </header>
    <main role="main">
      <%= render 'layouts/messages' %>
      <%= yield %>
    </main>
  </body>
</html>
```

We have the **viewport** metatag, a title, and a description.

We have partials for navigation links and flash messages.

Finally we have HTML5 structural elements.

That's all we need for now. In the next chapter, we'll revise it to support styling with Bootstrap.

Simple CSS

So far, we've examined four files that were added by the rails_layout gem:

- **app/views/layouts/application.html.erb**

- **app/views/layouts/_messages.html.erb**

- **app/views/layouts/_navigation.html.erb**

- **app/views/layouts/_navigation_links.html.erb**

Let's examine the CSS file that was created by the rails_layout gem.

Open the file **app/assets/stylesheets/simple.css**:

```css
/*
 * Simple CSS stylesheet for a navigation bar and flash messages.
 */
main {
  background-color: #eee;
  padding-bottom: 80px;
  width: 100%;
  }
header {
  border: 1px solid #d4d4d4;
  background-image: linear-gradient(to bottom, white, #f2f2f2);
  background-color: #f9f9f9;
  -webkit-box-shadow: 0 1px 10px rgba(0, 0, 0, 0.1);
  -moz-box-shadow: 0 1px 10px rgba(0, 0, 0, 0.1);
  box-shadow: 0 1px 10px rgba(0, 0, 0, 0.1);
  margin-bottom: 20px;
  font-family: 'Helvetica Neue', Helvetica, Arial, sans-serif;
}
ul.nav li {
  display: inline;
}
ul.nav li a {
  padding: 10px 15px 10px;
  color: #777777;
  text-decoration: none;
  text-shadow: 0 1px 0 white;
}
.flash_notice, .flash_alert {
  padding: 8px 35px 8px 14px;
  margin-bottom: 20px;
  text-shadow: 0 1px 0 rgba(255, 255, 255, 0.5);
  border: 1px solid #fbeed5;
  -webkit-border-radius: 4px;
  -moz-border-radius: 4px;
  border-radius: 4px;
  font-family: "Helvetica Neue", Helvetica, Arial, sans-serif;
  font-size: 14px;
  line-height: 20px;
}
.flash_notice {
  background-color: #dff0d8;
  border-color: #d6e9c6;
  color: #468847;
}
.flash_alert {
  background-color: #f2dede;
  border-color: #eed3d7;
```

```
    color: #b94a48;
}
```

If you already know CSS, you'll see we've set a background color for the `main` section of the page. We've also set styles for a header, navigation links, and flash messages. This book is about Rails, not CSS, so we won't examine this closely. For more on CSS, there are thousands of tutorials on the web, but I like these:

- Codeacademy

- HTML Dog

Remember what we learned about the Rails asset pipeline. By default, any CSS file in the **app/assets/stylesheets/** folder will be added automatically to the **application.css** file that is included in the default application layout.

In the next chapter, we remove the **app/assets/stylesheets/simple.css** and use Bootstrap to supply styles for the header, navigation links, and flash messages. For now, the **simple.css** file adds some basic styling and layout to the application.

Test the Application

Let's run the application to see how it looks with the new application layout. The web server may already be running. If not, enter the command:

```
$ rails server
```

Open a web browser window and navigate to http://localhost:3000/.

If you experimented with adding flash messages "Welcome" and "My birthday is soon," you'll see the messages when you visit the home page.

Our home page now has only one navigation link, for "Home." We'll add links for "About" and "Contact" pages soon.

Git

Let's commit our changes to the Git repository and push to GitHub:

```
$ git add -A
$ git commit -m "update application layout"
$ git push
```

Chapter 17

Front-End Framework

This chapter discusses front-end development and design using CSS. I'll show you how to add style to a Rails application, using Bootstrap for a simple theme.

What do we mean by "front-end development"? A website *back end* is the Rails application that assembles files that are sent to the browser, plus a database and any other server-side services. A website *front end* is all the code that runs in the browser. Everything that controls the appearance of the website in the browser is the responsibility of a front-end developer, including page layout, CSS stylesheets, and JavaScript code.

Front-end development has grown increasingly important as websites have become more sophisticated. And front-end technology has grown increasingly complex, to the degree that front-end development has become a job for specialists.

Front-end developers are primarily concerned with:

- markup - the layout and structure of the page

- style - graphic design for visual communication

- interactivity - browser-based visual effects and user interaction

Broader concerns include:

- cross-browser and cross-device functionality

- interaction design to improve website usability

- accessibility for users with physical or perceptual limitations

For years, front-end development was haphazard; webmasters each had their own quirky techniques. Around the time that Rails became popular, front-end developers at large companies began to share best practices and establish open source projects to bring structure and consistency to front-end development, leading to development of CSS frameworks.

CSS Frameworks

Web developers began putting together "boilerplate" CSS stylesheets as early as 2000, when browsers first began to fully support CSS. Boilerplate CSS made it easy to reuse CSS stylesheet rules from project to project. More importantly, designers often implemented "CSS reset" stylesheets to enforce typographic uniformity across different browsers.

Engineers at Yahoo! released the Yahoo! User Interface Library (YUI) as an open source project in February 2006. Inspired by an article by Jeff Croft, and reacting to the huge size of the YUI library, independent developers began releasing other CSS frameworks such as the 960 grid system and the Blueprint CSS framework.

There are dozens of CSS frameworks. In general, they all seek to implement a common set of requirements:

- An easily customizable grid

- Some default typography

- A typographic baseline

- CSS reset for default browser styles

- A stylesheet for printing

More recently, with the ubiquity of smartphones and tablets, CSS frameworks support responsive web design, accommodating differences in screen sizes across a range of devices.

In tandem with the development of CSS frameworks, we've seen the emergence of JavaScript libraries and frameworks.

JavaScript Libraries and Frameworks

JavaScript has nothing like RubyGems, a built-in package manager for code libraries, so initially there were few open source JavaScript libraries. Now there are several competing JavaScript package managers and many software libraries.

Prototype was one of the first open source JavaScript libraries, created by Sam Stephenson in February 2005 to improve JavaScript support in Ruby on Rails. MooTools, Dojo, and jQuery soon followed. Of these libraries, jQuery has become the most popular, largely because of thousands of modular jQuery *plug-ins* that implement a wide range of effects and *widgets* (web page features). These plug-ins are used to add visual effects and interactivity to web pages. Examples are drop-down menus, modal windows, tabbed panels, autocompletion search forms, and sliders or carousels for images. Even without plugins, jQuery is useful as a high-level interface for manipulating the browser DOM (document object model), to make it easy to do things like hiding or revealing HTML elements on a page. Any Rails application can use jQuery because it is included by default in any new Rails application.

Libraries such as jQuery add functionality to server-side applications, such as those built with Rails. Other JavaScript libraries serve as fully featured web

application development frameworks, allowing developers to build client-side applications that run in the browser and only interact with a server to read or write data. Examples of these full-fledged JavaScript frameworks are Ember.js, AngularJS, and Backbone.js. All use a variant of the model-view-controller (MVC) software design pattern to implement single-page applications which function more like desktop or mobile applications than websites. Developers who build a single-page application with one of these frameworks often use Ruby on Rails as a back end; an MVC JavaScript framework can replace all the Rails view files. None of these JavaScript frameworks dominate web application development like Ruby on Rails, but they are gaining popularity for single-page applications. We won't look at Ember.js, AngularJS, or Backbone.js in this book; they are an advanced topic and require entire books themselves.

The biggest problem with adding JavaScript to Rails is the difficult-to-maintain "JavaScript soup" that results from adding JavaScript to Rails views. Ember.js, AngularJS, or Backbone.js are more than is needed for simply structuring JavaScript in Rails views. But there's a newer framework that can be used to dry up JavaScript soup. It's React, a JavaScript framework developed by engineers at Facebook. Unlike AngularJS or Ember.js, React only manages views, not connections to databases or routing of requests, so it is not a full-stack framework, just a framework for the view layer. React's approach to building web pages is abstract and complex. But React is a good choice for complex interactive features, if you're determined to avoid JavaScript soup in your Rails application.

We won't look at JavaScript frameworks in this book, but we will use the Bootstrap framework to manage our CSS stylesheets.

Front-End Frameworks

Front-end frameworks combine CSS and JavaScript libraries. Many elements that are found on sophisticated web pages, such as modal windows or tabs,

require a combination of JavaScript and CSS. Combining CSS and JavaScript libraries in a common framework makes it possible to standardize and reuse common web page features.

There are many responsive front-end frameworks to choose from, including:

- Bootstrap

- Zurb Foundation

- Material Design Lite

- Bourbon Neat

- Semantic UI

- and many others

Each has its fans, though Bootstrap and Zurb Foundation are the most popular among Rails developers. Each adds a library of markup, styles, and standardized web page features such as modal windows, pagination, breadcrumbs, and navigation.

Bootstrap is the best-known front-end framework. It is the result of an effort to document and share common design patterns and assets across projects at Twitter, released as an open source project in August 2011.

Just ahead, we'll look at why we use Bootstrap in this book. But first, you'll need to learn about Sass.

CSS Preprocessing with Sass

Ordinary CSS is not a programming language. As a result, CSS rules are verbose and often repetitive. To add efficiency to CSS, Bootstrap relies on a CSS

preprocessor named Sass. Sass extends CSS to give it more powerful programming language features. As a result, your stylesheets can use variables, mixins, and nesting of CSS rules, just like a real programming language.

For example, in Sass you can create a variable such as `$blue:   #3bbfce` and specify colors anywhere using the variable, such as `border-color:   $blue`. *Mixins* are like variables that let you use snippets of reusable CSS. *Nesting* eliminates repetition by layering CSS selectors.

Sass is included in any new Rails application with the default sass-rails gem.

Bootstrap or Others?

Which front-end framework should you use? Bootstrap or another such as Zurb Foundation?

The Bootstrap team maintains a Ruby gem that provides a drop-in version of Bootstrap for Rails. Other front-end frameworks, such as Zurb Foundation, are also available as Ruby gems. But Bootstrap has a large developer community and many more third-party projects, as evidenced by a Big Badass List of Useful Twitter Bootstrap Resources. In its sheer magnitude, this list, from Michael Buckbee and Bootstrap Hero, demonstrates the popularity of Bootstrap and the vitality of its open source community. We'll use Bootstrap here because of its sheer popularity and solid support.

Before I show you how to integrate Bootstrap with your Rails application, let's briefly consider matters of design.

Graphic Design Options

There are three approaches to graphic design for your Rails application.

If you're well-funded and well-connected, you can put together a team or hire a freelance graphic designer to implement a unique design, built from scratch

using CSS or customized from a framework such as Bootstrap or Zurb Foundation. If you've got strong design skills, or can partner with an experienced web designer, you'll get a custom design that expresses the purpose and motif of your website.

A second approach is to use Bootstrap or another CSS framework to quickly add attractive CSS styling to your application. Many developers don't have the skill or resources to customize the design. Consequently, sites that use Bootstrap look very similar. If that's your situation, it's okay, really! It's better to have a decent site with the clean look of Bootstrap than to leak ugliness onto the web.

A third option is to obtain a pre-designed theme for your website. You may have visited ThemeForest or other theme galleries that offer pre-built themes for a few dollars each. These huge commercial galleries offer themes for Word-Press, Tumblr, or CMS applications such as Drupal or Joomla. They don't offer themes for Rails. Your best option is to convert open source themes designed with Bootstrap. You can visit sites such as Start Bootstrap, Bootswatch, or the Themestrap gallery to find Bootstrap themes.

Whether you use a prepackaged theme or design your own layout, you'll need to know how to set up a front-end framework in Rails. We'll look at setting up Bootstrap next.

Bootstrap 3 or 4?

At the time this was written, the new Bootstrap 4 version was in alpha release. The Bootstrap 4 version is significantly different from earlier versions. It has been in alpha for testing and feedback for over a year. Many Rails developers are already using the Bootstrap 4 version but almost all themes and third-party add-ons are only available for Bootstrap 3. Eventually, the Bootstrap 4 beta and final versions will be released but until then, I recommend sticking with Bootstrap 3. As a beginner, you'll find more resources and help for Bootstrap 3. We'll use it here for this tutorial.

Bootstrap Gem

Bootstrap provides a standard grid for layout plus dozens of reusable components for common page elements such as navigation, forms, and buttons. More importantly, it gives CSS the kind of structure and convention that makes Rails popular for back-end development. Bootstrap is packaged as a gem.

In your **Gemfile**, you've already added:

```
gem 'bootstrap-sass'
```

and previously run **$ bundle install**.

The bootstrap-sass gem provides the files required to integrate Bootstrap 3 with Rails. Developers who want to use the new Bootstrap 4 release will use the Bootstrap 4 gem.

Rather than following the installation instructions provided in the Bootstrap documentation, we'll use the rails_layout gem to set up Bootstrap and create the files we need.

Rails Layout Gem with Bootstrap

In the previous chapter, we used the rails_layout gem to configure the default application layout with HTML5 elements, navigation links, and flash messages. Now we'll use the rails_layout gem to set up Bootstrap and generate new files for the application layout as well as the navigation and messages partials. The new files will replace the layout files we created in the previous chapter.

We'll use the generator provided by the rails_layout gem to set up Bootstrap and add the necessary files. Run:

```
$ rails generate layout:install bootstrap3 --force
```

With the **--force** argument, the rails_layout gem will replace existing files. The gem will create the file:

- **app/assets/stylesheets/1st_load_framework.css.scss**

and modify the files:

- **app/assets/javascripts/application.js**
- **app/views/layouts/_messages.html.erb**
- **app/views/layouts/_navigation.html.erb**
- **app/views/layouts/application.html.erb**

It will also remove the file:

- **app/assets/stylesheets/simple.css**

Let's examine the files to see how our application is configured to use Bootstrap.

Renaming the application.css File

The rails_layout gem renamed the **app/assets/stylesheets/application.css** file as **app/assets/stylesheets/application.css.scss**. Note the **.scss** file extension. This will allow you to use the advantages of an improved syntax for your application stylesheet.

You learned earlier that stylesheets can use variables, mixins, and nesting of CSS rules when you use Sass.

Sass has two syntaxes. The most commonly used syntax is known as "SCSS" (for "Sassy CSS"), and is a superset of the CSS syntax. This means that every valid CSS stylesheet is valid SCSS as well. SCSS files use the extension **.scss**. The Sass project also offers a second, older syntax with indented formatting that uses the extension **.sass**. We'll use the SCSS syntax.

You can use Sass in any file by adding the file extension **.scss**. The asset pipeline will preprocess any **.scss** file and expand it as standard CSS.

For more on the advantages of Sass and how to use it, see the Sass website or the Sass Basics RailsCast from Ryan Bates.

Before you continue, make sure that the rails_layout gem renamed the **app/assets/stylesheet** file as **app/assets/stylesheets/application.css.scss**. Otherwise you won't see the CSS styling we will apply.

The application.css.scss File

In the previous chapter, I introduced the Rails *asset pipeline*.

Your CSS stylesheets get concatenated and compacted for delivery to the browser when you add them to this directory:

- **app/assets/stylesheets/**

The asset pipeline helps web pages display faster in the browser by combining all CSS files into a single file (it does the same for JavaScript).

Let's examine the file **app/assets/stylesheets/application.css.scss**:

```
/*
 * This is a manifest file that'll be compiled into application.css, which will include all the
 * listed below.
```

```
 *
 * Any CSS and SCSS file within this directory, lib/assets/stylesheets, vendor/assets/styleshee
 * or vendor/assets/stylesheets of plugins, if any, can be referenced here using a relative pat
 *
 * You're free to add application-wide styles to this file and they'll appear at the bottom of
 * compiled file so the styles you add here take precedence over styles defined in any styles
 * defined in the other CSS/SCSS files in this directory. It is generally better to create a ne
 * file per style scope.
 *
 *= require_tree .
 *= require_self
 */
```

The **app/assets/stylesheets/application.css.scss** file serves two purposes.

First, you can add any CSS rules to the file that you want to use anywhere on your website. Second, the file serves as a *manifest*, providing a list of files that should be concatenated and included in the single CSS file that is delivered to the browser.

If you are familiar with CSS syntax, it may seem odd that the relevant lines are commented out (using asterisks). These lines are not CSS, so they must be commented out so they won't be interpreted as CSS. Though they are commented out, the Rails asset pipeline reads and understands them. It's a bit of a hack, but it works.

A Global CSS File

Any CSS style rules that you add to the **app/assets/stylesheets/application.css.scss** file will be available to any view in the application. You could use this file for any style rules that are used on every page, particularly simple utility rules such as highlighting or resetting the appearance of links. However, in practice, you are more likely to modify the style rules provided by Bootstrap. These modifications don't belong in the **app/assets/stylesheets/application.css.scss** file; they will go in the **app/assets/stylesheets/1st_load_framework.css.scss** file.

In general, it's bad practice to place a lot of CSS in the **app/assets/stylesheets/application.cs** file (unless your CSS is very limited). Instead, structure your CSS in multiple

files. CSS that is used on only a single page can go in a file with a name
that matches the page. Or, if sections of the website share common elements,
such as themes for landing pages or administrative pages, make a file for each
theme. How you organize your CSS is up to you; the asset pipeline lets you
organize your CSS so it is easier to develop and maintain. Just add the files to
the **app/assets/stylesheets/** folder.

A Manifest File

It's not obvious from the name of the **app/assets/stylesheets/application.css.scss**
file that it serves as a *manifest file* as well as a location for miscellaneous CSS
rules. For most websites, you can ignore its role as a manifest file. In the com-
ments at the top of the file, the `*= require_self` directive indicates that any
CSS in the file should be delivered to the browser. The `*= require_tree .`
directive (note the Unix "dot operator") indicates any files in the same folder,
including files in subfolders, should be combined into a single file for delivery
to the browser.

If your website is large and complex, you can remove the `*= require_tree`
`.` directive and specify individual files to be included in the file that is gener-
ated by the asset pipeline. This gives you the option of reducing the size of the
application-wide CSS file that is delivered to the browser. For example, you
might segregate a file that includes CSS that is used only in the site's adminis-
trative section. In general, only large and complex sites need this optimization.
The speed of rendering a single large CSS file is faster than fetching multiple
files.

Bootstrap JavaScript

Bootstrap provides both CSS and JavaScript libraries.

Like the **application.css.scss** file, the **application.js** file is a manifest that al-
lows a developer to designate the JavaScript files that will be combined for

delivery to the browser.

The rails_layout gem modified the file **app/assets/javascripts/application.js** to include the Bootstrap JavaScript libraries:

```
//= require jquery
//= require jquery_ujs
//= require turbolinks
//= require bootstrap-sprockets
//= require_tree .
```

It added the directive `//= require bootstrap-sprockets` before `//= require_tree .`.

Bootstrap CSS

The rails_layout gem added a file **app/assets/stylesheets/1st_load_framework.css.scss** containing:

```
// import the CSS framework
@import "bootstrap-sprockets";
@import "bootstrap";
.
.
.
```

The file **app/assets/stylesheets/1st_load_framework.css.scss** is automatically included and compiled into your Rails application.css file by the `*= require_tree .` statement in the **app/assets/stylesheets/application.css.scss** file. The file could be named anything. However, by giving it a name beginning with "1" it will load before any other stylesheet files we may add later. The asset pipeline loads files in alphabetical order. We want the Bootstrap framework to load before any custom CSS files.

The `@import "bootstrap";` and `@import "bootstrap-sprockets";` directives will import the Bootstrap CSS rules from the Bootstrap gem.

You could add the Bootstrap `@import` code to the **app/assets/stylesheets/application.css.sc** file. However, it is better to have a separate **app/assets/stylesheets/1st_load_framework.cs** file. You may wish to modify the Bootstrap CSS rules; placing changes to Bootstrap CSS rules in the **1st_load_framework.css.scss** file will keep your CSS better organized.

In addition to the simple `@import` directives, the **app/assets/stylesheets/1st_load_framewo** contains a collection of Sass mixins. We'll look at these later in the chapter.

Using Bootstrap CSS Classes

Now that you've installed Bootstrap, you have a rich library of interactive effects you can add to your pages.

Take a look at the Bootstrap documentation to see your options. Here are just a few examples:

- buttons

- navigation bar

- alerts

At a simpler level, Bootstrap provides a collection of carefully-crafted styling rules in the form of CSS classes. These are building blocks you use for page layout and typographic styling. For example, Bootstrap gives you CSS classes to set up rows and columns in a grid system.

Let's take a closer look at the Bootstrap grid system.

Bootstrap Grid

The Bootstrap grid is responsive because it has "breakpoints." There are actually four grids:

- Extra small: browser windows 0 to 768 pixels wide (phones)

- Small: browser windows 768 to 992 pixels wide (tablets)

- Medium: browser windows 992 to 1200 pixels wide (desktops)

- Large: browser windows 1200 pixels and wider (large desktops)

Start by designing for the extra small screen; then add classes prefixed "small," "medium," or "large" if you want a different layout for larger screens. The layout will change at each breakpoint.

The grid gives you 12 columns by default. You can organize your layout in horizontal and vertical sections using **row** and **columns** classes.

For example, you could use Bootstrap grid classes to set up an application layout with a footer as a row with two sections:

```html
<div class="container">
  <footer class="row">
    <section class="col-xs-4">
      Copyright 2016
    </section>
    <section class="col-xs-8">
      All rights reserved.
    </section>
  </footer>
</div>
```

The Bootstrap **row** class will create a horizontal break. The footer will contain two side-by-side sections. The first will be four columns wide; the second will be eight columns wide.

Here's the same footer with a responsive design:

```html
<div class="container">
  <footer class="row">
    <section class="col-xs-12 col-sm-4">
      Copyright 2016
    </section>
```

```
    <section class="col-xs-12 col-sm-8">
      All rights reserved.
    </section>
  </footer>
</div>
```

On desktops and tablets, the footer will contain two side-by-side sections. On phones, each section will expand to take the full browser width, appearing as stacked rows.

To better understand the grid system with all its options, see the documentation for the Bootstrap grid.

Presentational Versus Semantic Styles

There are two schools of thought among front-end developers. Some developers are content to use Bootstrap's classes directly in Rails view files. For these developers, the Bootstrap classes are both practical and descriptive, making it easy for any developer who knows the Bootstrap framework to visualize the layout of a page.

Other developers take issue with this approach. They argue that Bootstrap's markup is often *presentational*, with class names describing the appearance of the page. In an ideal world, all markup would be *semantic*, with class names describing the function or purpose of a style. For example, a submit button often needs styling. Compare these two approaches to markup:

- presentational: `<button class="big red button">Order Now</button>`

- semantic: `<button class="submit">Order Now</button>`

Suppose your user testing indicates a green button generates more sales. With the presentational approach you'd have to change both the Rails view file and the CSS file. With a semantic approach, you'd just change the CSS file to reassign the color of the `submit` class.

Use Bootstrap Classes Directly

For quick and simple websites, where you don't need to be concerned about long-term maintenance, use Bootstrap's CSS classes directly.

For example, you can style a button like this:

- `<button type="button" class="btn btn-success">Order Now</butto`

The `btn-success` class is semantic and applies an "alert color" which is green, by default, in Bootstrap.

Or Use Sass Mixins with Bootstrap

You can use Sass mixins to create your own semantic class names.

Sass mixins add a layer of complexity that can map Bootstrap class names to your own semantic class names.

For example, the Bootstrap grid system is presentational. Specifying rows and columns, and quantifying the size of columns, describes the visual appearance of sections of the layout rather than the purpose of each section. The presentational approach makes it easy to visualize the layout of a page. But you'll be tied to Bootstrap 3 class names for the life of your website. If class names change in a future version of Bootstrao, or you decide to switch to another front-end framework, it will be difficult to update your application, as you will have to carefully rebuild each view file.

Is it worth the effort to add the complexity of Sass mixins just to future-proof your website? Probably not for a simple website such as the one you are building for Foobar Kadigan.

The rails_layout gem uses Sass mixins to apply CSS style rules to the default application layout. In doing so, the default application layout is free of framework-specific code and can be used with either Bootstrap or Zurb Foundation.

Before we examine the default application layout, let's take a look at the Sass mixins supplied by the rails_layout gem.

Look again at the file **app/assets/stylesheets/1st_load_framework.css.scss** created by the rails_layout gem:

```scss
// import the CSS framework
@import "bootstrap-sprockets";
@import "bootstrap";

// make all images responsive by default
img {
  @extend .img-responsive;
  margin: 0 auto;
  }
// override for the 'Home' navigation link
.navbar-brand {
  font-size: inherit;
  }

// THESE ARE EXAMPLES YOU CAN MODIFY
// create your own classes
// to make views framework-neutral
.column {
  @extend .col-md-6;
  @extend .text-center;
  }
.form {
  @extend .col-md-6;
  }
.form-centered {
  @extend .col-md-6;
  @extend .text-center;
  }
.submit {
  @extend .btn;
  @extend .btn-primary;
  @extend .btn-lg;
  }
// apply styles to HTML elements
// to make views framework-neutral
main {
  @extend .container;
  background-color: #eee;
  padding-bottom: 80px;
  width: 100%;
  margin-top: 51px; // accommodate the navbar
  }
section {
  @extend .row;
```

```
  margin-top: 20px;
  }
```

The rails_layout gem is in active development so the file you've created may be different from the example in this tutorial. It will probably be very similar.

At the top of the file we import the Bootstrap framework CSS files from the gem.

We override a Bootstrap style rule so the "Home" navigation link matches the other links in the navigation bar.

Then we use mixins to create semantic classes.

Mixins can take a block of CSS styles, other mixins, or a CSS selector (a CSS class or ID).

If you'd like to combine CSS classes, or rename a CSS class, use the **@extend** directive.

The first declaration **column** combines the Bootstrap classes **col-md-6** and **text-center** to make a new class, **column**.

Next we create a few classes that combine Bootstrap CSS classes. For example, the new **submit** class can be used for a button. When we use it in a view, this class will be purely semantic since it describes the purpose of the element, allowing us to set its appearance outside of any view file.

Finally, to avoid applying Bootstrap classes in the application layout file, we apply styles to HTML elements **main** and **section** to make the views framework-neutral. We use the **@extend** directive to add a Bootstrap CSS class. And we directly set standard CSS properties such as **background-color** and **margin-top**.

Using this technique, the file **app/assets/stylesheets/1st_load_framework.css.scss** becomes the single point of intersection between the Bootstrap framework and the application layout. For a simple website, this could be over-engineering and counter-productive. The rails_layout gem uses the technique so that either Bootstrap or Zurb Foundation can be used without any change to the default application layout.

We'll use the CSS classes provided by the rails_layout gem in the tutorial application, but if you choose to customize the application, feel free to use Bootstrap classes directly to keep your project simple.

Application Layout with Bootstrap

Let's look at the application layout file created by the rails_layout gem:

Examine the contents of the file **app/views/layouts/application.html.erb**:

```erb
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title><%= content_for?(:title) ? yield(:title) : "Learn Rails" %></title>
    <meta name="description"
      content="<%= content_for?(:description) ? yield(:description) : "Learn Rails" %>">
    <%= stylesheet_link_tag 'application', media: 'all', 'data-turbolinks-track' => 'reload' %>
    <%= javascript_include_tag 'application', 'data-turbolinks-track' => 'reload' %>
    <%= csrf_meta_tags %>
  </head>
  <body>
    <header>
      <%= render 'layouts/navigation' %>
    </header>
    <main role="main">
      <%= render 'layouts/messages' %>
      <%= yield %>
    </main>
  </body>
</html>
```

This file is almost identical to the simple application layout file we looked at in the previous chapter.

Because we've applied Bootstrap classes to the HTML element `main` in the **app/assets/stylesheets/1st_load_framework.css.scss** file, there's no need to use Bootstrap classes directly in the application layout.

Flash Messages with Bootstrap

The messages partial we use with Bootstrap is complex.

Examine the file **app/views/layouts/_messages.html.erb**:

```erb
<%# Rails flash messages styled for Bootstrap 3.0 %>
<% flash.each do |name, msg| %>
  <% if msg.is_a?(String) %>
    <div class="alert alert-dismissible
        alert-<%= name.to_s == 'notice' ? 'success' : 'danger' %>">
      <button type="button" class="close" data-dismiss="alert"
        aria-hidden="true">&times;</button>
      <%= content_tag :div, msg, :id => "flash_#{name}" %>
    </div>
  <% end %>
<% end %>
```

We use **each** to iterate through the flash hash, retrieving a **name** and **msg** that are passed to a block to be output as a string. The expression **if msg.is_a?(String)** serves as a test to make sure we only display messages that are strings.

We construct a div that applies Bootstrap CSS styling around the message. Bootstrap provides classes **alert** and **alert-dismissible** to style the message.

We use the Ruby ternary operator to check the type of the alert. A class of either **success** or **danger** styles the message. Rails **notice** messages will get styled with the Bootstrap **success** class. Any other Rails messages, including **alert** messages, will get styled with the Bootstrap **danger** class.

We use the Rails **content_tag** view helper to create a div containing the message.

Bootstrap creates a "close" icon by applying the class **alert-dismissible**. Bootstrap's integrated JavaScript library will hide the alert box when the "close" link is clicked.

Bootstrap provides detailed documentation if you want to change the styling of the alert boxes.

Navigation Partial with Bootstrap

The layout and styling required for the Bootstrap navigation bar are in the navigation partial file.

Examine the file **app/views/layouts/_navigation.html.erb**:

```erb
<%# navigation styled for Bootstrap 3.0 %>
<nav class="navbar navbar-inverse navbar-fixed-top">
  <div class="container">
    <div class="navbar-header">
      <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-c
        <span class="sr-only">Toggle navigation</span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </button>
      <%= link_to 'Home', root_path, class: 'navbar-brand' %>
    </div>
    <div class="collapse navbar-collapse">
      <ul class="nav navbar-nav">
        <%= render 'layouts/navigation_links' %>
      </ul>
    </div>
  </div>
</nav>
```

The navigation partial is now more complex, with layout and Bootstrap classes needed to produce a responsive navigation bar.

At the conclusion of this chapter, you'll test the responsive navigation by resizing the window. At small sizes, the navigation links will disappear and be replaced by an icon labeled "Menu." Clicking the icon will reveal a vertical menu of navigation links. The navigation menu is a great demonstration of the ability of Bootstrap to adjust to the small screen size of a tablet or smartphone.

If you'd like to add a site name or logo to the tutorial application, you can replace the link helper `<%= link_to 'Home', root_path %>`. It is important to preserve the enclosing layout and classes, even if you don't want to display a site name or logo. The enclosing layout is used to generate the navigation menu when the browser window shrinks to accommodate a tablet or

smartphone.

You'll see we wrap the nested partial **render 'layouts/navigation_links'** with Bootstrap classes to complete the navigation bar.

Navigation Links Partial

The file **app/views/layouts/_navigation_links.html.erb** is unchanged:

```erb
<%# add navigation links to this file %>
```

Later we'll add links to "About" and "Contact" pages.

The navigation links partial will be simply a list of navigation links.

We're following the *separation of concerns* principle here. By separating the links from the styling that creates the navigation bar, we segregate the code that is unique to Bootstrap. In the future, if the Bootstrap layout or CSS classes change, we can make changes without touching the navigation links. If we wish, we can replace the navigation partial and substitute one that uses a different framework instead of Bootstrap, leaving the navigation links intact.

Set up SimpleForm with Bootstrap

One of the requirements for our tutorial application is a contact form. We could set up styling for the form when we implement the contact page, but it is convenient to set up form styling now, as we would if we were adding multiple forms to the site.

Rails provides a set of view helpers for forms. They are described in the RailsGuides: Rails Form Helpers document. But, as you've learned, Rails has more than one stack, and most developers use an alternative set of form

helpers named SimpleForm, provided by the SimpleForm gem. The Simple-Form helpers are more powerful, easier to use, and offer an option for styling with Bootstrap.

In your **Gemfile**, you've already added:

```
gem 'simple_form'
```

and previously run **$ bundle install**.

Run the generator to install SimpleForm with a Bootstrap option:

```
$ rails generate simple_form:install --bootstrap
```

which installs several configuration files:

```
config/initializers/simple_form.rb
config/initializers/simple_form_bootstrap.rb
config/locales/simple_form.en.yml
lib/templates/erb/scaffold/_form.html.erb
```

Here the SimpleForm gem uses the **rails generate** command to create files for initialization and localization (language translation). SimpleForm can be customized with settings in the initialization file. We'll use the defaults.

Test the Application

Let's see how the application looks with Bootstrap. The web server may already be running. If not, enter the command:

```
$ rails server
```

Open a web browser window and navigate to http://localhost:3000/.

You should see a new page design that displays Bootstrap styling. Thanks to the open source efforts of the Bootstrap core team and contributors, we've added powerful front-end features to our website with little effort.

You can click the "X" close icons to hide the flash messages, thanks to the integrated CSS and JavaScript of the Bootstrap framework.

Next we'll add "About" and "Contact" pages to the application. After we update the navigation links, you'll see how the Bootstrap responsive web design adjusts the navigation bar at different browser widths.

Remove the Flash Messages

Before we continue, we'll remove the flash messages we created for our demonstration.

Update the file **app/controllers/visitors_controller.rb**:

```ruby
class VisitorsController < ApplicationController

  def new
    @owner = Owner.new
  end

end
```

Git

Let's commit our changes to the Git repository and push to GitHub:

```
$ git add -A
$ git commit -m "front-end framework"
$ git push
```

Chapter 18

Add Pages

Let's begin adding pages to our web application.

There are three types of web pages in a Rails application. We've looked at two types so far:

- static pages in the **public/** folder that contain no Ruby code

- dynamic pages such as our home page that use the application layout

There's another type of web page that is required on many websites. It has static content; that is, no dynamic data is needed on the page. But it uses the default application layout to maintain consistency in the website look and feel. We classify this type of page as a:

- static view that uses the application layout

Examples include:

- "About" page

- Legal page

- FAQ page

It's possible to place these pages in the **public/** folder and copy the HTML and CSS from the default application layout but this leads to duplicated code and maintenance headaches. And dynamic elements such as navigation links can't be included. For these reasons, developers seldom create static pages in the **public/** folder.

Alternatively, a dynamic page can be created that has no model, a nearly-empty controller, and a view that contains no instance variables. This solution is quite common for static views that use the application layout.

This solution is implemented so frequently that many developers create a gem to encapsulate the functionality. We're going to use the best-known of these gems, the high_voltage gem created by the Thoughtbot consulting firm.

We'll use the High Voltage gem to create an "About" Page.

We also will create a Contact page. We'll again use the High Voltage gem, but only for the first version of the Contact page. Later we'll discard the page we created with the High Voltage gem and replace it with a full model-view-controller implementation. The process will show the difference between an older form of web application architecture and a newer "Rails way."

High Voltage Gem

We can add a page using the High Voltage gem almost effortlessly. The gem implements Rails "convention over configuration" so well that there is nothing to configure. There are alternatives to its defaults which can be useful but we won't need them; visit the GitHub home page for the high_voltage gem if you want to explore all the options.

In your **Gemfile**, you've already added:

```
gem 'high_voltage'
```

and previously run **$ bundle install**.

Views Folder

Create a folder **app/views/pages**:

```
$ mkdir app/views/pages
```

Any view files we add to this directory will automatically use the default application layout and appear when we use a URL that contains the filename.

The High Voltage gem contains all the controller and routing magic required for this to happen.

Let's try it out.

"About" Page

Create a file **app/views/pages/about.html.erb**:

```erb
<% content_for :title do %>About<% end %>
<h3>About Foobar Kadigan</h3>
<p>He was born in Waikikamukau, New Zealand. He left New Zealand for England,
  excelled at the University of Mopery, and served in the Royal Loamshire Regiment.
  While in service, he invented the kanuten valve used in the processing of
  unobtainium for industrial use. With a partner, Granda Fairbook, he founded
  Acme Manufacturing, later acquired by the Advent Corporation, to develop his
  discovery for use in the Turboencabulator. Mr. Kadigan is now retired and
  lives in Middlehampton with a favorite cat, where he raises Griadium frieda
  and collects metasyntactic variables.</p>
<p>His favorite quotation is:</p>
<p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod
  tempor incididunt ut labore et dolore magna aliqua.</p>
```

Our simple "About" view will be combined with the default application layout by the High Voltage gem.

We include a **`content_for`** Rails view helper that passes a page title to the application layout.

Contact Page

For the initial version of the Contact page, create a file **app/views/pages/contact.html.erb**:

```erb
<% content_for :title do %>Contact<% end %>
<h3>Contact</h3>
```

This is a placeholder page we'll use to test a navigation link.

We include a **`content_for`** Rails view helper that passes a page title to the application layout.

Routing for the High Voltage Gem

The High Voltage gem provides a PagesController. You'll never see it; it is packaged inside the gem.

In addition to providing a controller, the High Voltage gem provides default routing so any URL with the form http://localhost:3000/pages/about will obtain a view from the **app/views/pages** directory.

Like the PagesController, the code that sets up the route is packaged inside the gem. For details about the syntax of routing directives, refer to RailsGuides: Routing from the Outside In.

Update the Navigation Partial

You can use a Rails route helper to create a link to any view in the **app/views/pages** directory like this:

```
link_to 'About', page_path('about')
```

Let's add links to the "About" and "Contact" pages.

Replace the contents of the file **app/views/layouts/_navigation_links.html.erb** with this:

```erb
<%# add navigation links to this file %>
<li><%= link_to 'About', page_path('about') %></li>
<li><%= link_to 'Contact', page_path('contact') %></li>
```

With an updated navigation bar, we can test the application.

Test the Application

The web server may already be running. If not, enter the command:

```
$ rails server
```

Open a web browser window and navigate to http://localhost:3000/.

Links to the pages "About" and "Contact" should work.

If you get an error "uninitialized constant PagesController," make sure the **config/routes.rb** file looks like this:

```ruby
Rails.application.routes.draw do
  root to: 'visitors#new'
end
```

Watch what happens when you resize the page. At smaller sizes, the navigation bar changes to display a menu icon. Clicking the menu icon reveals a drop-down menu of navigation links. You're seeing the power of the Bootstrap framework.

Here's a troubleshooting tip. If clicking the menu icon doesn't reveal a drop-down menu, the application may not be loading the Bootstrap JavaScript library. Make sure that the file **app/assets/javascripts/application.js** contains:

```
//= require jquery
//= require jquery_ujs
//= require turbolinks
//= require bootstrap-sprockets
//= require_tree .
```

Git

Let's commit our changes to the Git repository and push to GitHub:

```
$ git add -A
$ git commit -m "add 'about' and 'contact' pages"
$ git push
```

There is nothing more we need for our "About" page.

In the next chapter, we'll explore two different implementations for the Contact page.

Chapter 19

Contact Form

Forms are ubiquitous on the web, to the degree we seldom notice how often they are used for data entry, whether we're logging into a website or posting a blog comment. To build any interactive website, you'll need to understand forms. Here we'll build a contact form for our tutorial application.

A contact form is common on many websites. If you think about it, contact forms are often unnecessary; simply displaying an email address is sufficient, more convenient, and easier to implement. But building a contact form is an excellent way to learn how to handle user data input. We'll pretend that our odd client, Mr. Foobar Kadigan, insists that he needs a contact form on his website.

We're not backing the tutorial application with a database so we won't store the contact data after the information is submitted. Instead, in a subsequent chapter we'll learn how to send the contents of the form by email to the website owner.

The "Old Way" and the "Rails Way"

In this chapter, we'll explore two ways to implement a contact form. The first way will be familiar to anyone who has used PHP or similar web platforms. It is an obvious and straightforward way to handle a form. As we look closer,

we'll see the approach has limitations. We'll discard our first approach and rebuild the Contact page, discovering how the "Rails way" is more powerful.

You may wonder why I'm going to show you two different ways to implement the contact form.

First, it is worthwhile to see there is more than one way to implement a web application. Maturity as a software developer means imagining different approaches and evaluating your options. With this exercise, you'll contrast two approaches and see how we make choices about software architecture.

More importantly, it is not always obvious why we do things in a "Rails way." It would be easy to simply walk you through the steps to build a contact form without showing you alternative implementations (that's how most tutorials do it). But you'll gain a deeper understanding of Rails by building the contact form in a less sophisticated fashion and then seeing the more elegant Rails approach.

User Story

Let's plan our work with a user story:

```
*Contact Page*
As a visitor to the website
I want to fill out a form with my name, email address, and some text
In order to send a message to the owner of the website
```

Our first step will be to create a route to a controller that will process the submitted form.

Routing

We're going to create a ContactsController to process the submitted form data. Every form must have a destination URL that receives the form submission.

We need to set a route to generate the destination URL.

Open the file **config/routes.rb**. Replace the contents with this:

```ruby
Rails.application.routes.draw do
  post 'contact', to: 'contacts#process_form'
  root to: 'visitors#new'
end
```

The route **post 'contact', to: 'contacts#process_form'** will create a route helper that generates a URL and hands off the request to a controller.

You can run the **rails routes** command to see our routes in the console:

```
$ rails routes
 Prefix Verb URI Pattern         Controller#Action
contact POST /contact(.:format) contacts#process_form
   root GET  /                   visitors#new
   page GET  /pages/*id          high_voltage/pages#show
```

The output of the **rails routes** command is somewhat cryptic but confirms we've created the routes we need.

The first item in the **rails routes** output indicates we can add "contact" to "_path" to get our route helper, **contact_path**:

- **contact_path** - a route helper that can be used in a controller or view

The second item indicates the request will be handled with the HTTP POST protocol:

- **POST** - HTTP method to submit form data

The third item indicates the application will respond to the following URL:

- http://localhost:3000/contact - URL generated by the route helper

The fourth item indicates a request to the URL will be handled by:

- **contacts** - the name of the controller

- **process_form** - a controller action

For details about the syntax of routing directives, refer to RailsGuides: Routing from the Outside In.

The route won't work yet; we need to create a ContactsController. But first we'll create the form.

Adding a Form to the Contact Page

You'll recall that we set up the SimpleForm gem when we added Bootstrap to our application. The SimpleForm gem provides Rails view helpers for forms. You'll remember that we described Rails view helpers as "macros to generate HTML." The SimpleForm gem gives us view helpers to generate all the HTML required by complex forms. Forms require some of the most complex HTML a developer will encounter, so the SimpleForm gem is truly worthwhile.

Let's add the code for a contact form to the Contact page.

Replace the contents of the file **app/views/pages/contact.html.erb**:

```erb
<% content_for :title do %>Contact<% end %>
<h3>Contact</h3>
<div class="form">
  <%= simple_form_for :contact, url: contact_path do |form| %>
    <%= form.input :name, autofocus: true %>
    <%= form.input :email %>
    <%= form.input :content, as: :text %>
    <%= form.button :submit, 'Submit', class: 'submit' %>
  <% end %>
</div>
```

The code is compact but complex. We see several elements:

- **content_for** is a view helper that passes a page title to the application layout

- **<div class="form">** sets the width of the form and applies any styles we desire

- **simple_form_for** is the view helper for the form

The **simple_form_for** view helper instantiates a form object which we assign to a variable named **form**. SimpleForm offers many standard form elements, such as text fields and submit buttons. Each element is available as a method call on the form object.

The view helper **simple_form_for** requires *parameters* and a *block*.

Every form needs a name and a route in the application that will handle processing of the form data. The parameters are:

- **contact** - the name of the form

- **url** - set to **contact_path**, the destination for the form data

Later, when we change this form to accommodate the"Rails way," we'll replace these two parameters with a single instance variable. The magic of Rails will generate the name of the form and the destination URL from the instance variable. For now, to implement the "old way," we supply the name of the form and the destination URL.

The **simple_form_for** view helper accommodates a Ruby block. The block begins with **do** and closes with **end**. The code inside the block works just like code inside a method. In this case, the **form** object is passed to the block and methods belonging to the **form** object are called to produce HTML output.

Inside the block, the **form** object methods generate HTML for:

- a name field

- an email field

- a content field

- a submit button

Each of the form methods takes various parameters, such as:

- **autofocus** - displays the cursor in the field

- **as: :text** - displays a multiline text area

- **input_html** - adds any HTML such as a CSS class

- **class** - applies a CSS class to modify a button's appearance

The structure of the form is clearly visible in the code. The form begins with a **simple_form_for** helper and closes with the **end** keyword. Each line of code produces an element in the form such as a field or a button.

This is a common structure for a Rails view helper and it will soon become familiar.

Controller

We need code to process the form data. The form data is sent to the server as a POST request attached to a URL. As we've learned, in Rails we use controllers to respond to browser requests. For this implementation, we'll create a ContactsController to process the submitted form data.

Create a file **app/controllers/contacts_controller.rb**:

```
class ContactsController < ApplicationController

  def process_form
    Rails.logger.debug "DEBUG: params are #{params.inspect}"
    flash[:notice] = "Received request from #{params[:contact][:name]}"
    redirect_to root_path
  end

end
```

The ContactsController inherits the behavior of the base ApplicationController.

We create a **process_form** method to respond when the form is submitted. Later we'll learn that **process_form** doesn't fit the "Rails way." We'll use it for now.

Before we look closely at the code for the **process_form** method, we need to learn about the **params** hash.

Params Hash

Take a close look at these two lines:

```
    Rails.logger.debug "DEBUG: params are #{params.inspect}"
    flash[:notice] = "Received request from #{params[:contact][:name]}"
```

Notice the **params** object.

Earlier we learned about the Ruby *Hash* class. It is a data structure for key/value pairs and Hash instances are ideal for storing form data. Each field on the form can be mapped as *label* and *data*, or key and value, and stored in a Hash.

Rails does all the work of extracting the form data from the browser's POST request. Rails creates a hash with the form field data mapped to the form field labels and gives the hash the name of the form. Here's the hash as pure Ruby code:

```
contact = {name: 'Daniel', email: 'daniel@danielkehoe.com', content: 'hi!'}
```

Rails goes a step further and nests the form hash inside another hash named **params**.

As pure Ruby code, the **params** hash looks like this:

```
params = {controller: 'contacts',
          action: 'process_form',
          contact: {name: 'Daniel', email: 'daniel@danielkehoe.com', content: 'hi!'}
          }
```

The **params** hash includes these elements (plus others we won't cover):

- current controller

- current action

- form data (our **contact** hash)

You will see the contents of the **params** hash in the console log after you submit the form. We'll look at the console log when we test the implementation.

Process_form Method

Now that we know about the **params** hash, take a look again at the **process_form** method:

```
def process_form
  Rails.logger.debug "DEBUG: params are #{params.inspect}"
  flash[:notice] = "Received request from #{params[:contact][:name]}"
  redirect_to root_path
end
```

We use a `logger.debug` method to reveal the form data in our console log by revealing the contents of the `params` hash. The `inspect` method shows the parameters in an easy-to-read list.

Then we extract the data posted to the name field of the form and construct a flash message. A hash containing the data from the contact form is nested inside the `params` hash. We can retrieve the value of the name field with the expression `params[:contact][:name]`. We use double quotes and string interpolation to form the message using the `#{...}` syntax that evaluates a Ruby expression and combines it with a string.

Finally we use the `redirect_to` directive to render the home page.

We haven't actually sent the contact data to anyone. We'll add code for that later, after we refactor the controller to be a better example of the "Rails way." Before we do that, let's test the current implementation. We've already set up routing for the new controller.

Test the Application

If you need to start the server:

```
$ rails server
```

Open a web browser window and navigate to http://localhost:3000/.

Click the "Contact" link; then fill out and submit the form.

You should see the flash message "Received request from ..." on the home page. If you see the message "My birthday is soon" you need to delete your earlier experiment from the Visitors controller.

If you get an error message "uninitialized constant ContactsController," make sure you created the file **app/controllers/contacts_controller.rb** as instructed above.

Notice what appears in the console log:

```
Started POST "/contact" for ::1 at 2015-12-24 13:46:34 +0700
Processing by ContactsController#process_form as HTML
Parameters: {"utf8"=>"", "authenticity_token"=>"...",
  "contact"=>{"name"=>"Daniel Kehoe", "email"=>"daniel@danielkehoe.com",
  "content"=>"Looking forward to your birthday!"}, "commit"=>"Submit"}
DEBUG: params are <ActionController::Parameters {"utf8"=>"",
  "authenticity_token"=>"...", "contact"=>{"name"=>"Daniel Kehoe",
  "email"=>"kehoe@fortuity.com", "content"=>"hi"}, "commit"=>"Submit",
  "controller"=>"contacts", "action"=>"process_form"} permitted: false>
Redirected to http://localhost:3000/
Completed 302 Found in 15ms (ActiveRecord: 0.0ms)
```

The console log is our most important tool for debugging. Let's analyze what
we see:

- *Started POST* - shows the server is responding to an HTTP POST request

- *"/contact"* - the path portion of the URL

- *at . . .* - timestamp

- *Processing by ContactsController* - the controller

- *process_form* - the controller action (the method that handles the request)

- *as HTML* - not XML or some other markup

- *Parameters:* - the `params` hash containing all the submitted data

- *"utf8"=""* - a Rails workaround to set the language encoding in Internet
  Explorer

- *"authenticity token"* - prevents CSRF security exploits

- *"contact"* - a hash containing the form data

- *"commit"* - the "Submit" label from the button

- *DEBUG* - our debug message containing the form data

- *Redirected to http://localhost:3000/* - responded by displaying the home page

- *Completed 302 Found* - HTTP response status code 302 indicating a redirection

- *in 0ms* - time required to process the request

That's a lot of data. For now, we really only care about the form data buried in the **params** hash.

You can see that we really don't need the debug message because the console log shows us the contents of the **params** hash.

The Validation Problem

It looks like we've got everything we need to handle a form submission. As a next step, we could implement code to send an email message using form data extracted from the **params** hash.

But consider a potential problem. What if the email address is poorly formed? The visitor will think the message has been sent but it will never be delivered.

Or what if the name field or message is blank? It's not just a problem for the hapless visitor. An evildoer could repeatedly click the submit button, filling Foobar Kadigan's email inbox with endless empty messages.

We need *validation* of the form data before we process it.

We could dig into the Rails **String** API and look for a way to test if the string is empty or contains only whitespaces. And we could raise an **Exception** if the string is blank.

Here's what validation code could look like. We won't use this code (because there's a better way to do this):

```
class ContactsController < ApplicationController

  def process_form
    if params[:contact][:name].blank?
      raise 'Name is blank!'
    end
    if params[:contact][:email].blank?
      raise 'Email is blank!'
    end
    if params[:contact][:content].blank?
      raise 'Message is blank!'
    end
    message = "Received request from #{params[:contact][:name]}"
    redirect_to root_path, :notice => message
  end

end
```

We would need additional code to test for invalid email addresses (it will be a complex *regex*, or *regular expression*). And we would need a nicer way of showing the error to the visitor (right now, raising the exception displays an error message that makes it appear the application is broken). If we were implementing this on another web application platform, we might go further down this path, googling for code examples, and implementing a lengthy but bulletproof validation function.

Rails offers a better way.

Remove the Contact Page

We will implement a model-view-controller architecture for our Contact feature. That means we need a Contact model, a Contacts controller, and view files in the **app/views/contacts/** folder.

We no longer need the Contact page in the **app/views/pages/** folder.

Let's get started by removing the file **app/views/pages/contact.html.erb**:

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

Before we implement a model-view-controller architecture, let's take time to understand the advantages of the Rails model-view-controller approach.

Implementing the "Rails Way"

Our initial implementation of the contact form is consistent with the earliest approach to web application development. That's why I call it the "old way." It is an approach that originated in 1993 with a specification for CGI, the Common Gateway Interface. Before CGI, every page on the web existed only as a static HTML file. CGI made it possible to run a program, or CGI script, that dynamically generated HTML. In the early years of the web, every web URL matched either an HTML file or a CGI script. This is the "page paradigm" of the web.

So far, we're following the "page paradigm." Our Contact page hosts the form. Clicking the submit button makes a request to another page that is actually a program that returns HTML. Until the late 1990s, this is how the web worked. But soon after the introduction of CGI, developers began exploring the possibility of running a single program (an application server) that responds to any URL, parsing the URL to establish routing, and generating pages dynamically. This was the genesis of the "web application paradigm." It's how Rails works.

The web application paradigm frees us from one-to-one correspondence of a URL with a single file or script. It allows us to refactor our code into object-oriented classes and methods that can be inherited rather than duplicated, which means we don't repeat the same code on every page that processes a form.

The web application paradigm makes it possible to use the model-view-controller architecture. Instead of looking at the web as URLs that return pages, we see requests that are routed to controllers that render views. We can segregate any code that manipulates data into a model class, instead of mixing HTML with

data manipulation in a single script. With the "web application paradigm," we can have a generic model class that isolates the code that connects to a database or validates form data. We can create models that inherit the generic behavior from a parent class and get a database connection or validation "for free." Unlike the "page paradigm," we'll avoid duplicating validation code every time we need to process a form.

Consider our **process_form** method again:

```ruby
class ContactsController < ApplicationController

  def process_form
    if params[:contact][:name].blank?
      raise 'Name is blank!'
    end
    if params[:contact][:email].blank?
      raise 'Email is blank!'
    end
    if params[:contact][:content].blank?
      raise 'Message is blank!'
    end
    message = "Received request from #{params[:contact][:name]}"
    redirect_to root_path, :notice => message
  end

end
```

We'll replace it with something better.

Our "segregation of concerns" philosophy suggests that validation belongs in a model, since validation is a type of data manipulation (strictly speaking, a test of data integrity). Furthermore, it would be nice to make the validation tests generic so they could be used to validate data submitted from any form.

Rails, as a framework, provides all this for us. We call it the "Rails way."

ActiveModel

Rails extracts and generalizes common code that every website requires. The code that websites need for access to databases is abstracted into the Rails

ActiveRecord class. ActiveRecord includes code from the ActiveModel class that handles interaction with forms and data validation.

The ActiveModel class interfaces with SimpleForm to provide sophisticated validation and error handling. We can mix in behavior from the ActiveModel class to add validation and error handling to any model we create.

SimpleForm will recognize ActiveModel methods if we provide a model as an argument to the SimpleForm view helper. SimpleForm will give the form a name that matches the model name. And SimpleForm will automatically generate a destination URL for the form based on the model name.

More significantly, SimpleForm will add sophisticated error handling to the form. If a visitor doesn't enter a name or submits an invalid email address, and we declare in our model that we require validation, SimpleForm will highlight the invalid field and display an inline message indicating the problem. Compared to what we've implemented so far, this kind of error handling provides a vastly superior user experience. Instead of displaying a message that the application failed, the form will be redisplayed with the problem marked and noted.

Now that we've seen the advantages of the "Rails way," let's re-implement our contact form using the model-view-controller architecture.

Model

When we build database-backed applications with Rails, we base our models on a parent class named ActiveRecord. We are not using a database for our tutorial application, so we'll mix in behavior from ActiveModel, which adds validation and error handling to our model. Let's set up a model that doesn't require a database.

Create a file **app/models/contact.rb**:

```ruby
class Contact
  include ActiveModel::Model
  attr_accessor :name, :string
```

```
  attr_accessor :email, :string
  attr_accessor :content, :string

  validates_presence_of :name
  validates_presence_of :email
  validates_presence_of :content
  validates_format_of :email,
    with: /\A[-a-z0-9_+\.]+\@([-a-z0-9]+\.)+[a-z0-9]{2,4}\z/i
  validates_length_of :content, :maximum => 500

end
```

When you copy this, be careful to keep the long regex expression (**with: /.../i**) on one line (no line breaks).

We give the model the name "Contact."

We mix in behavior from the ActiveModel class using **include ActiveModel::Model**.

We create attributes (data fields) for the model by using the **attr_accessor** keyword, specifying that each attribute is a string. The attributes match the fields in the contact form.

ActiveModel gives us validation methods named **validates_presence_of**, **validates_format_of**, and **validates_length_of**. We check that **name**, **email**, and **content** exist (no blanks are allowed). We provide a complex *regex*, or *regular expression*, to test if the email address is valid. Finally, we declare that the message content cannot exceed 500 characters.

The model is elegant. We describe the fields we need and state our validation requirements. ActiveModel does all the rest.

Next we'll add a new Contact page by creating a view in the **app/views/contacts/** folder.

The new contact form will use our new model.

Create a New Contact Page

First, let's create the **app/views/contacts/** folder:

```
$ mkdir app/views/contacts/
```

Create a file **app/views/contacts/new.html.erb**:

```erb
<% content_for :title do %>Contact<% end %>
<h3>Contact</h3>
<div class="form">
  <%= simple_form_for @contact do |form| %>
    <%= form.error_notification %>
    <%= form.input :name, autofocus: true %>
    <%= form.input :email %>
    <%= form.input :content, as: :text %>
    <%= form.button :submit, 'Submit', class: 'submit' %>
  <% end %>
</div>
```

The form is the same as we used before, but we're now providing only one argument, the **@contact** instance variable, to the SimpleForm view helper. That's enough to generate the form name and destination URL.

We haven't yet created a controller that assigns the Contact model to the **@contact** instance variable. We'll do that soon.

Earlier, I explained that SimpleForm configures itself if we provide a model that inherits from ActiveRecord. SimpleForm gives the form a name that matches the model name. And SimpleForm generates a destination URL based on the model name.

SimpleForm uses the **@contact** instance variable to name the form, set a destination for the form data, and initialize each field in the form using attributes from the Contact model. Setting the values for the form fields from the attributes in the model is called "binding the form to the object" and you can read about it in the RailsGuides: Form Helpers article.

We've added the `error_notification` method which provides all the error handling. The method call is very simple but the results will be impressive.

We'll need a controller and routing to complete our model-view-controller architecture. But first, we'll detour to learn about seven standard controller actions.

Seven Controller Actions

Consider all the possibilities for managing a list. It's a list of anything: users, inventory, thingamajigs. We use a web application to manage the list, so we'll fill out a form to record each item in our list.

The web application offers seven features to help us manage our records:

- ***index*** - display a list of all items

- ***show*** - display a record of one item

- ***new*** - display an empty form

- ***create*** - save a record of a new item

- ***edit*** - display a record for editing

- ***update*** - save an edited record

- ***destroy*** - delete a record

You can manage any list using these seven actions. There are a few extra actions that are helpful, such as:

- *pagination* - displaying a portion of a list

- *sorting* - displaying the list in a different order

- *bulk edit* - changing multiple items at once

But seven basic actions are all you need for managing any list of items.

The "Rails way" is about taking advantage of structure and convention to leverage the power of the framework.

The ApplicationController contains code to implement each of the seven standard actions. When we create a controller that inherits from the Application-Controller, we get these standard actions "for free." That's why our **new** method in our VisitorsController was so simple. The controller knew to render a view file named **new.html.erb** from the **views/visitors/** folder because of behavior inherited from the ApplicationController.

Just like the Rails directory structure provides consistency to make it easy for any Rails developer to collaborate with other Rails developers, relying on the seven standard controller actions makes it easy for other team members to understand how your controllers work.

A controller that uses these actions is said to be "RESTful" (a term that refers to representational state transfer, a software design abstraction). Experienced Rails developers follow the "Rails way" and try to use RESTful controller methods when possible.

When necessary, you will add other controller actions. For example, imagine you've built a subscription website. When a user's subscription ends, you may not want to **destroy** the subscriber record. Instead you might add a controller **expire** or **suspend** action that marks the subscriber record as expired so you can continue to access the subscriber's contact information for customer service or renewal offers. To the extent you can, use the seven standard controller actions and be cautious about adding more.

Earlier, I said our ContactsController **process_form** method isn't suitable for the "Rails way." With our model-view-controller architecture, we can piggyback on the ApplicationController to display our empty contact form and process the form when it is submitted.

We'll use only two of the seven standard controller actions:

- *new* - display the empty contact form

- *create* - validate and process the submitted form

Our ContactsController will know to render a view from the **app/views/contacts/new.html.e** file when we call the controller `new` method.

We won't piggyback on behavior from the ApplicationController `create` method. But we'll implement a `create` method because, by convention, the form will submit the data to the controller's `create` method. SimpleForm will create a destination URL that corresponds to the ContactsController#create action.

Controller

Replace the contents of the file **app/controllers/contacts_controller.rb**:

```ruby
class ContactsController < ApplicationController

  def new
    @contact = Contact.new
  end

  def create
    @contact = Contact.new(secure_params)
    if @contact.valid?
      # TODO send message
      flash[:notice] = "Message sent from #{@contact.name}."
      redirect_to root_path
    else
      render :new
    end
  end

  private

  def secure_params
    params.require(:contact).permit(:name, :email, :content)
  end

end
```

We've dropped the "old school" `process_form` method and added the "Rails way" `new` and `create` methods.

The controller `new` action will instantiate an empty Contact model, assign it to the `@contact` instance variable, and render the **app/views/contacts/new.html.erb** view. We've already created the view file containing the form.

SimpleForm will set a destination URL that corresponds to the ContactsController#create action. The `create` method will instantiate a new Contact model using the data from the form (we take steps to avoid security vulnerabilities first—more on that later).

The ActiveModel class provides a method `valid?` which we can call on the Contact model. Our conditional statement `if @contact.valid?` checks each of the validation requirements we've set in the model.

If all the Contact fields are valid, we can send a message (which we'll add later), prepare a flash message, and redirect to the home page. Notice that we don't need to dig into the `params` hash for the visitor's name; it is now available as `@contact.name` directly from the model.

If any validation fails, the controller `create` action will render the **app/views/contacts/new.** view. This time, appropriate error messages are set and the form object's `error_notification` method will highlight the invalid field and display a matching prompt.

You're looking at the tightly bound interaction of the "Rails way" model, view, and controller.

The only element we are missing is routing. But first, let's look closer at the steps we take to avoid security exploits.

Mass-Assignment Vulnerabilities

Rails protects us from a class of security exploits called "mass-assignment vulnerabilities." Rails won't let us initialize a model with just any parameters submitted on a form. Suppose we were creating a new user and one of the user

attributes was a flag allowing administrator access. A malicious hacker could create a fake form that provides a user name and sets the administrator status to "true." Rails forces us to "white list" each of the parameters used to initialize the model.

We create a method named **secure_params** to screen the parameters sent from the browser. The **params** hash contains two useful methods we use for our screening:

- **require(:contact)** - makes sure that **params[:contact]** is present

- **permit(:name, :email, :content)** - our "white list"

With this code, we make sure that **params[:contact]** only contains **:name, :email, :content**. If other parameters are present, they are stripped out. Rails will raise an error if a controller attempts to pass params to a model method without explicitly permitting attributes via **permit**.

In older versions of Rails (before Rails 4.0), the mass-assignment exploit was blocked by using a "white list" of acceptable parameters with the **attr_accessible** keyword in a model. You'll see this code in examples and tutorials that were written before Rails 4.0 introduced "strong parameters" in the controller.

Private Methods

If you paid close attention to the code you added to the Contacts controller, you may have noticed the keyword **private** above the **secure_params** method definition. This is a bit of software architecture that limits access to the **secure_params** method (plus any more methods we might add beneath it).

Very simply, adding the **private** keyword restricts access to the **secure_params** method so only methods in the same class can use it. You might be puzzled; after all, how else could it be accessed? We haven't explored calling methods from other classes, so I'll just say that without the **private** keyword, the

secure_params method could be used from code anywhere in our application. In this case, we apply the **private** keyword because we want to be sure the **secure_params** method is only used in the **ContactsController** class. It's just a bit of "best practice" and for now, you can simply learn that **secure_params** method should be a private method.

Now let's look at routing for controllers that are built the "Rails way."

Routing

Rails routing is aware of the seven standard controller actions.

In fact, it takes only one keyword (with one parameter) to generate seven different routes for any controller.

The keyword is **resources** and supplying a name that matches a model and controller provides all seven routes.

Open the file **config/routes.rb**. Replace the contents with this:

```
Rails.application.routes.draw do
  resources :contacts, only: [:new, :create]
  root to: 'visitors#new'
end
```

Here we've added **resources :contacts, only: [:new, :create]**.

We only want two routes so we've added the restriction **only: [:new, :create]**.

The **new** route has these properties:

- **new_contact_path** - route helper

- **contacts** - name of the controller (ContactsController)

- **new** - controller action

- http://localhost:3000/contacts/new - URL generated by the route helper

- **GET** - HTTP method to display a page

The **create** route has these properties:

- **contacts_path** - route helper

- **contacts** - name of the controller (ContactsController)

- **create** - controller action

- http://localhost:3000/contacts - URL generated by the route helper

- **POST** - HTTP method to submit form data

You can run the **rails routes** command to see these in the console:

```
$ rails routes
     Prefix Verb URI Pattern             Controller#Action
   contacts POST /contacts(.:format)     contacts#create
new_contact GET  /contacts/new(.:format) contacts#new
       root GET  /                       visitors#new
       page GET  /pages/*id              high_voltage/pages#show
```

The output of the **rails routes** command shows we've created the routes we need.

Our new route **new_contact_path** can now be used. We've completed our move to the model-view-controller architecture by adding the appropriate routes.

Change Navigation Links

With our new model-view-controller architecture, we need to change the navigation links.

Change the file **app/views/layouts/_navigation_links.html.erb**:

```
<%# add navigation links to this file %>
<li><%= link_to 'About', page_path('about') %></li>
<li><%= link_to 'Contact', new_contact_path %></li>
```

We're ready to test the model-view-controller implementation of the Contact feature.

Be sure you've removed the file **app/views/pages/contact.html.erb**, as it is no longer used.

Test the Application

If you need to restart the server:

```
$ rails server
```

Open a web browser window and navigate to http://localhost:3000/.

Click the "Contact" link; then fill out and submit the form.

You should see the flash message "Message sent from . . . " on the home page.

Try submitting the form with a blank name. You'll see a warning message, "Please review the problems below."

Try submitting the form with an invalid email address such as "me@foo". The form will re-display with a message, "Please review the problems below," and next to the email field, "is invalid."

Combining SimpleForm error handling with ActiveModel validation is powerful. If validation fails after the form is submitted, the page will redisplay and SimpleForm will display an appropriate error message.

Git

Let's commit our changes to the Git repository and push to GitHub:

```
$ git add -A
$ git commit -m "contact form"
$ git push
```

We've built a sophisticated Contact form.

Chapter 20

Send Mail

Email sent from a web application is called transactional email. As a website visitor, you've probably seen transactional email such as these messages:

- sign up confirmation email

- response to a password reset request

- acknowledgment of a purchase

- notice of a change to a user profile setting

A web application can send email to a visitor. It can also send messages to its owner or webmaster. On large active sites, email notices can be impractical (an admin interface is better) but for our small-volume tutorial application, it makes sense to email the contact request directly to the site owner (Foobar Kadigan is retired and enjoys receiving email).

User Story

Let's plan our work with a user story:

```
*Send Contact Message*
As the owner of the website
I want to receive email messages with a visitor's name, email address, and some text
In order to communicate with visitors
```

To implement the user story, let's create a feature that sends the contact data as an email message.

Implementation

Rails makes it easy to send email. The ActionMailer gem is part of any Rails installation.

Implementation of email closely follows the model-view-controller architecture. To implement email, you'll need:

- model

- view

- mailer

The "mailer" is similar to a controller, combining data attributes from a model with a view file. Any methods we add to the mailer class can be called from a controller, triggering delivery of an email message.

The model can be any we've already created. In this case, we'll use the Contact model, since it gives us access to the visitor's name, email address, and message.

We'll create a mail-specific view file in the **app/views/user_mailer/** folder. Our folder for mail-specific views will go in the **app/views/** directory as a sibling of the **app/views/layouts** folder.

The Rails directory structure already gives us a folder **app/mailers/** for the mailer class and, not surprisingly, it is a sibling of the **app/controllers/** folder.

We don't have to create the necessary folders and files manually, as the **rails generate** command runs a utility to create what we need.

Create View Folder and Mailer

Use the **rails generate** command to create a mailer with a folder for views:

```
$ rails generate mailer UserMailer
```

The name of the mailer isn't important; we'll use **UserMailer** because it is obvious.

The **rails generate** command will create a file:

- **app/mailers/user_mailer.rb**

It also creates test files which we won't use in this tutorial.

It uses three additional files which are provided by default in a Rails 5 application:

- **app/mailers/application_mailer.rb**

- **app/views/layouts/mailer.html.erb**

- **app/views/layouts/mailer.text.erb**

This implements our model-view-mailer architecture.

Edit the Mailer

Add a **contact_email** method to the mailer by editing the file **app/mailers/user_mailer.r**

```ruby
class UserMailer < ApplicationMailer
  default from: "do-not-reply@example.com"

  def contact_email(contact)
    @contact = contact
    mail(to: Rails.application.secrets.owner_email, from: @contact.email, :subject => "Website
  end
end
```

The **UserMailer** class inherits behavior from the **ApplicationMailer** class. We'll create a method definition that assigns the **contact** argument to the instance variable **@contact**. Like a controller that combines a model with a view, our mailer class makes the instance variable available in the view.

The name of the method isn't important; it can be anything obvious. We'll use it in the ContactsController to trigger mail delivery.

Like the **render** method in a web page controller, the ActionMailer parent class has a **mail** method that renders the view.

You'll need to use your email address in the mailer. You should have already set a configuration variable for your email address in the file **config/secrets.yml**. If you haven't done so, do it now. By inserting the configuration variable with your email address after **to:**, your inbox will receive the message. If Foobar Kadigan was a real person, we'd supply his email address here.

We need to insert a "from" address in two places. First there is a default, for all messages that do not set a "from" address. We will use "do-not-reply@example.com" for the default "from" address. The email is originating from a web application that does not receive email, so this indicates the email address should not be used for replies. For emails going to website visitors, it would be best to provide a default email address for a customer service representative on the "from" line, so the recipient can easily reply. We're not sending email messages to visitors so we can ignore this nicety.

For our `contact_email` method, we'll insert the email address of the visitor as the "from" address since we are sending a message to the site owner. This makes it easy for Foobar Kadigan to click "reply" when he is reading the contact messages in his inbox. You can see our use of the email attribute from the Contact model in the expression `from:    @contact.email`.

That's all we need for mailer class. Next we'll create a view containing the message.

Create Mailer View

There are two types of mailer views. One contains plain text, for recipients who don't like formatted email (some people still read email from the Unix command line). The other type contains HTML markup to provide formatting. It's good to create a message of both types, though most recipients will benefit from HTML formatting.

The mailer view for formatted email looks very similar to a web page view file. It contains HTML markup plus Ruby expressions embedded in `<%= ...  %>` delimiters. In the `UserMailer` class, we've assigned the Contact model to the instance variable `@contact` so any attributes are available for use in the message.

Create a file **app/views/user_mailer/contact_email.html.erb**:

```
<!DOCTYPE html>
<html>
  <head>
    <meta content="text/html; charset=UTF-8" http-equiv="Content-Type" />
  </head>
  <body>
    <h1>Website Contact</h1>
    <p>
      This visitor requested contact:
    </p>
    <p>
      <%= @contact.name %><br/>
      <%= @contact.email %><br/>
```

```
    </p>
    <p>
      The visitor said:
    </p>
    <p>
      "<%= @contact.content %>"
    </p>
  </body>
</html>
```

You can easily imagine how this view would look as a web page. You'll soon see it as an email message in your inbox.

For those recipients who like plain text, create a view without HTML markup.

Create a file **app/views/user_mailer/contact_email.text.erb**:

```
You received a message from <%= @contact.name %> with email address <%= @contact.email %>.

The visitor said:

"<%= @contact.content %>"
```

You've created views for the email message.

Now we can integrate our email feature with the ContactsController.

Modify Controller

We'll add code to the ContactsController:

```
UserMailer.contact_email(@contact).deliver_now
```

Replace the contents of the file **app/controllers/contacts_controller.rb**:

```ruby
class ContactsController < ApplicationController

  def new
    @contact = Contact.new
  end

  def create
    @contact = Contact.new(secure_params)
    if @contact.valid?
      UserMailer.contact_email(@contact).deliver_now
      flash[:notice] = "Message sent from #{@contact.name}."
      redirect_to root_path
    else
      render :new
    end
  end

  private

  def secure_params
    params.require(:contact).permit(:name, :email, :content)
  end

end
```

The **UserMailer** class is available to any controller in the application. We call the **contact_email** method we've created, passing the **@contact** instance variable as an argument, which renders the email message. Finally, the **deliver_now** method initiates delivery.

For more on sending email from a Rails application, see RailsGuides: Action Mailer Basics.

Test the Application

If your web server is not running, start it:

```
$ rails server
```

Open a web browser window and navigate to http://localhost:3000/.

Click the "Contact" link and try submitting the form.

The email message should be visible in the console.

If you didn't get an email message in your inbox, make sure you set your **config/environments/development.rb** file to perform deliveries as described in the "Configuration" chapter. Be sure to restart your server if you change the configuration file.

Troubleshooting

If you get an error, you can practice troubleshooting. You've set up a complex system with many dependencies. It's great if it works, but there are several opportunities for errors.

The most likely errors are a missing user name or password. We are trying to connect to the SendGrid email service. SendGrid expects your SendGrid user name. Mine is **DanielKehoe** (it is not my email address). SendGrid also expects a password.

User Name Issues

If you get the error message, "SMTP-AUTH requested but missing user name," SendGrid is not receiving a user name it recognizes.

Check that the user name is set in your Unix environment variables:

```
$ echo "$SENDGRID_USERNAME"
```

You should see your SendGrid user name.

Make sure you've used underscores consistently. If your Unix environment variable is **SENDGRID_USER_NAME** and the **config/secrets.yml** file contains **SENDGRID_USERNAME**, you'll have a problem.

Password Issues

If you get the error message, "SMTP-AUTH requested but missing secret phrase," SendGrid is not receiving the SendGrid password.

Check that the SendGrid password is set in your Unix environment variables:

```
$ echo "$SENDGRID_PASSWORD"
```

You should see the long cryptic string in the console response. Again, make sure you've used underscores consistently, and **SENDGRID_PASSWORD** is used for the Unix environment variable as well as the **config/secrets.yml** file.

Problems with Environment Variables

First, close and reopen your terminal to make sure your environment reflects any recent changes you've made to your shell configuration. Then try **echo "$SENDGRID_USERNAME"** to see if you get the credentials you set in your **.bash_profile** or **.bashrc** files.

If you've set up Unix environment variables but **echo "$SENDGRID_USERNAME"** doesn't return the correct variable in the console, you may have a problem with the way you've set Unix environment variables. Most computers use the bash shell and you can set environment variables in your **.bash_profile** or **.bashrc** files. But not every system is alike. If it seems Unix environment variables are not working, you may have to find a colleague who can help you troubleshoot.

If your Unix environment variables are not working, you can hardcode the variables in your **config/secrets.yml** file:

Replace the following:

```
development:
  email_provider_username: <%= ENV["SENDGRID_USERNAME"] %>
  email_provider_password: <%= ENV["SENDGRID_PASSWORD"] %>
```

with:

```
development:
  email_provider_username: example
  email_provider_password: 's#cr*t'
```

In a YAML file, you do not need quotes unless your string contains special characters. If your password contains any of these characters you should surround the string with single quotes:

```
: { } [ ] & * # ? | - < > = ! % @ \
```

Remember the security rule: Don't commit the **config/secrets.yml** file to Git if it contains any secrets. Test the application and finish your troubleshooting. Then remove the hardcoded values from the **config/secrets.yml** file before committing to Git.

Asynchronous Mailing

You may notice a delay in the responsiveness of the Contact form after adding the email feature. Unfortunately, there's a performance penalty with our new feature. Our controller code connects to the SendGrid server and waits for a response before it renders the home page and displays the acknowledgment message.

The performance penalty can be avoided by changing the implementation so that the controller doesn't wait for a response from the SendGrid server. We call this *asynchronous* behavior because sending email does not need to be "in sync" with displaying the acknowledgment. Eliminating a delay improves the user experience and makes the site feel more responsive. Asynchronous mailing requires a *queueing system* for *background jobs*.

For our tutorial application, and for a typical small business website, the delay caused by lack of queueing is no big deal. Keep in mind, though, as you tackle bigger projects in Rails, you will need to implement a queueing system. Rails includes the Active Job feature for background processing. The Mailing List with Active Job tutorial in the Capstone Rails Tutorials series explains how to use it.

Git

Let's commit our changes to the Git repository and push to GitHub:

```
$ git add -A
$ git commit -m "sending mail"
$ git push
```

You've created a Rails application that handles a form and sends email to the site owner.

Mail is a practical way to connect with site visitors. Let's implement a feature that collects email addresses for mass mailing of a newsletter.

Chapter 21

Mailing List

Even as other messaging avenues become increasingly popular, such as messaging apps or Facebook messages, email remains the most practical way to stay in touch with website visitors. Encouraging a visitor to provide an email address means offering an invitation to a dialog and a relationship beyond a single visit.

If you have a legitimate reason to stay in touch, and you've motivated the visitor to leave an email address, you'll need a mailing list service. You've seen how Rails can send an email message. From what you've seen so far, you can imagine it would not take much code to loop through a list of email addresses from a database, sending a message to each. In the early days of the web, it was easy for any system administrator to write a script for mass mailings. Since there is negligible cost to sending bulk email, unscrupulous and ignorant operators sent email to any address they could scrape, borrow, or steal. The resulting flood of spam made checking one's inbox an icky experience and destroyed much of the early culture of the Internet. Fortunately, services such as Gmail arose to filter email. There is now a thick (but leaky) layer of screening protocols that redirect spam to a junk folder. One reason you won't use a Rails application to send bulk email is that a web application server is not the most efficient tool for sending email. More significantly, there's a good chance your email won't go through or, if it does (and someone complains), you'll quickly see your IP

address blacklisted. That's why we use mailing list services to send bulk email such as newsletters or promotional offers.

Considerable expertise is required to keep email from being filtered as spam (see MailChimp's article Email Delivery For IT Professionals. Email service providers increase reliability of delivery. These services track deliveries and show how well your email is being delivered. You'll also get features such as management of "unsubscribe" requests and templates to design attractive messages.

There are at least a dozen well-established email service providers that allow a Rails application to programmatically connect to the service (via an API) to add or remove email addresses. For a list, see the article Send Email with Rails. For this tutorial application, we'll use MailChimp because there is no cost to open an account and you can send up to 12,000 emails/month to list of 2000 or fewer subscribers for free.

Spam is unsolicited email. Don't ever send spam, whether for yourself, a client, or an employer. If recipients complain, your IP address and domain name will be blacklisted. So be very careful to only send to subscribers who signed up, send what subscribers expect, and be sure to offer value. If you get complaints, or the unsubscribe rate is high, stop.

We'll assume we've discussed the rules with Foobar Kadigan and he is eager to offer a newsletter to his visitors that will be genuinely appreciated.

User Story

Let's plan our work with a user story:

```
*Subscribe to Mailing List*
As a visitor to the website
I want to sign up for a mailing list
In order to receive news and announcements
```

To implement the user story, we'll add a mailing list feature.

Implementation

We'll use the Rails model-view-controller architecture. We'll need:

- Visitors model

- view for visitors#new

- Visitors controller with **new** and **create** methods

- routing for visitors#new and visitors#create

We'll add a Visitor model that has a data attribute for an email address. We already have a Visitors controller that renders the home page using the file in the **app/views/visitors/** folder. We'll replace the contents of the view file with a nice photo, a marketing message, and a form.

Our Visitors controller **new** and **create** methods will be very similar to what we created for the Contacts controller. Instead of connecting to SendGrid to send a message, we'll call a method to save the visitor's email address to a MailChimp mailing list.

Gibbon Gem

The Gibbon gem is a convenient wrapper for the MailChimp API. We could connect to the MailChimp API using other gems that provide low-level plumbing such as HTTP connections (httparty) and data parsing (multi_json), but other developers have already done the work of wrapping the plumbing in a higher-level abstraction that easily fits into a Rails application. Amro Mousa's Gibbon gem is popular and actively maintained.

In your **Gemfile**, you've already added:

```
gem 'gibbon'
```

and previously run **$ bundle install**.

Home Page

Earlier we built a home page that provided a simple demonstration of the Ruby language. We'll discard it and replace it with a page that you could adapt for a typical small-business website.

We want a nice photo, space for a marketing message, and the "sign up" form.

Replace the contents of the file **app/views/visitors/new.html.erb**:

```erb
<% content_for :title do %>Foobar Kadigan<% end %>
<% content_for :description do %>Website of Foobar Kadigan<% end %>
<section>
  <img src="http://lorempixel.com/1170/600/cats/1">
</section>
<section>
  <div class="column">
    <h3>Stay in touch.</h3>
  </div>
  <div class="column">
    <div class="form-centered">
      <%= simple_form_for @visitor do |f| %>
        <%= f.error_notification %>
        <%= f.input :email, label: false, :placeholder => 'Your email address...' %>
        <br/>
        <%= f.button :submit, "Sign up for the newsletter", :class => "submit" %>
      <% end %>
    </div>
  </div>
</section>
```

We include `content_for` view helpers that pass a title and description to the application layout.

We add a photo to the page with an `<img>` tag. We're taking a shortcut and using a placeholder photo from the lorempixel.com service.

The `section` and `<div class="column">` tags apply our CSS grid to create a row with two columns, one for our marketing message, and one for the form.

Our marketing message is merely a placeholder. For a real website, you'd likely craft a stronger call to action than merely "Stay in touch."

The form is very similar to the form on the Contact page, except we initialize it with the `@visitor` instance variable and only need a field for an email address. We suppress display of the email field label with the flag `label:  false` and use the `:placeholder` parameter to create a hint in the empty input field.

A submit element will contain the text, "Sign up for the newsletter," and we apply a CSS class to style the element as a button.

Photo Options

You're free to modify this page as you wish, as long as you keep the form intact.

You might wish to modify the placeholder photo. If you don't like cats, try http://lorempixel.com/1170/600/nightlife/1 or any other categories from the lorempixel.com service. You can change the size by modifying the dimensions from 1170 (pixel width) by 600 (pixel height).

You can replace the placeholder photo with your own. Look for the **app/assets/images** folder and add an image. Instead of the HTML `<img>` tag, use the Rails `image_tag` view helper, like this:

```
<%= image_tag "myphoto.jpg" %>
```

We'll need a Visitor model to initialize the form.

Visitor Model

The Visitor model is almost identical to the Contact model we created earlier, except there is just one data attribute for the email field.

We'll also add a **subscribe** method to add a visitor to a MailChimp list. We'll call this method from the controller when we process the submitted form.

Create a file **app/models/visitor.rb**:

```ruby
class Visitor
  include ActiveModel::Model
  attr_accessor :email, :string
  validates_presence_of :email
  validates_format_of :email, with: /\A[-a-z0-9_+\.]+\@([-a-z0-9]+\.)+[a-z0-9]{2,4}\z/i

  def subscribe
    mailchimp = Gibbon::Request.new(api_key: Rails.application.secrets.mailchimp_api_key)
    list_id = Rails.application.secrets.mailchimp_list_id
    result = mailchimp.lists(list_id).members.create(
      body: {
        email_address: self.email,
        status: 'subscribed'
      })
    Rails.logger.info("Subscribed #{self.email} to MailChimp") if result
  end

end
```

When you copy this, be careful to keep the long regex expression **with:** **/.../i** on one line (no line breaks).

Just as we did for the Contact model, we use **include ActiveModel::Model** to mix in behavior from the ActiveModel class. This is the best way to create a model that does not use a database. In other applications, where models use a database, you will create a model class that inherits from ActiveRecord instead.

We create the email attribute using the **attr_accessor** keyword, specifying the email attribute will be a **string**. We set validation requirements using **validates_presence_of** and **validates_format_of** keywords.

To subscribe a visitor to a mailing list, you need to provide:

- list_id - identify the MailChimp list

- email_address - address of the visitor

- status - subscribed, pending, or unsubscribed

We specify "subscribed" to immediately add the address without asking the user for confirmation. We could specify "Pending" if we wanted to add the address with double-opt-in so the visitor receives a request to confirm the email address before they are subscribed.

Our `subscribe` method does the work of connecting to the MailChimp server to add the visitor to the mailing list. We instantiate the Gibbon object which provides all the connectivity, providing the `mailchimp_api_key` value, which we've set in the **config/secrets.yml** file. We assign the Gibbon object to the `mailchimp` variable (we could name it anything).

We set the `list_id` from a configuration variable in the **config/secrets.yml** file.

The visitor's `email_address` is an attribute of the model we obtain from `self`.

We must set the `status` property to `'subscribed'` to indicate the visitor should receive mailings.

Finally, if the application successfully adds the new subscriber, we write a message to the logger. If we get an error when trying to add the subscriber, Gibbon will raise an exception.

Visitors Controller

We already have a Visitors controller that contains a simple `new` method. We'll change the `new` method, add a `create` method, and provide a `secure_params` private method to secure the controller from mass assignment exploits.

Replace the contents of the file **app/controllers/visitors_controller.rb**:

```ruby
class VisitorsController < ApplicationController

  def new
    @visitor = Visitor.new
  end

  def create
    @visitor = Visitor.new(secure_params)
    if @visitor.valid?
      @visitor.subscribe
      flash[:notice] = "Signed up #{@visitor.email}."
      redirect_to root_path
    else
      render :new
    end
  end

  private

  def secure_params
    params.require(:visitor).permit(:email)
  end

end
```

Our **new** method now assigns the Visitor model to an instance variable instead of the Owner model.

The **create** method is almost identical to the Contacts controller **create** method. We instantiate the Visitor model with scrubbed parameters from the submitted form.

If the validation check succeeds, we subscribe the visitor to the MailChimp mailing list with the **@visitor.subscribe** method. All the work of connecting to MailChimp happens in the Visitor model.

If the validation check fails, we redisplay the home page (the **new** action).

Clean Up

We no longer use the Owner model, so we can delete the file **app/models/owner.rb**:

```
$ rm app/models/owner.rb
```

There's no harm if it remains but it is good practice to remove code that is no longer used.

Routing

Our routing is now more complex. In addition to rendering the visitors# new view as the application root (the home page), we need to handle the **create** action. We can use a "resourceful route" as we did with the Contacts controller.

Open the file **config/routes.rb**. Replace the contents with this:

```
Rails.application.routes.draw do
  resources :contacts, only: [:new, :create]
  resources :visitors, only: [:new, :create]
  root to: 'visitors#new'
end
```

The root path remains **visitors#new**. Order is significant in the **config/routes.rb** file. As the final designated route, the root path will only be active if nothing above it matches the route.

We've added **resources :visitors, only: [:new, :create]**.

We only want two routes so we've added the restriction **only: [:new, :create]**.

The **new** route has these properties:

- **new_visitor_path** - route helper

- **visitors** - name of the controller (VisitorsController)

- **new** - controller action

- http://localhost:3000/visitors/new - URL generated by the route helper

- **GET** - HTTP method to display a page

The **create** route has these properties:

- **visitors_path** - route helper

- **visitors** - name of the controller (VisitorsController)

- **create** - controller action

- http://localhost:3000/visitors - URL generated by the route helper

- **POST** - HTTP method to submit form data

You can run the **rails routes** command to see these in the console:

```
$ rails routes
      Prefix Verb URI Pattern              Controller#Action
    contacts POST /contacts(.:format)      contacts#create
new_contact GET  /contacts/new(.:format) contacts#new
    visitors POST /visitors(.:format)      visitors#create
new_visitor GET  /visitors/new(.:format) visitors#new
        root GET  /                        visitors#new
        page GET  /pages/*id               high_voltage/pages#show
```

The output of the **rails routes** command shows we've created the routes
we need.

Test the Application

If you need to start the server:

```
$ rails server
```

Open a web browser window and navigate to http://localhost:3000/.

You'll see our new home page with the placeholder photo and the "sign up" form.

Enter your email address and click the "sign up" button. You should see the page redisplay with an acknowledgment message. Try entering an invalid email address such as "me@foo@", or click the submit button without entering an email address, and you should see an error message.

You'll have to log in to MailChimp and check your list of subscribers to see if the new email address was added successfully.

With MailChimp, you can send a welcome message automatically when the visitor signs up for the mailing list. Use the welcome message to inform the visitor that they've successfully subscribed to the mailing list and will receive the next newsletter email.

It's a bit difficult to find the MailChimp option to create a welcome message. Strangely, MailChimp considers a welcome message a "form." Here's how to find it. On the MailChimp "Lists" page, click the "down arrow" for a list and click "Signup forms." Then click "General forms." On the "Create Forms" page, there is a drop-down list of "Forms & Response Emails." The gray box shows "Signup form." Click the down arrow. Select the menu item named "Final 'Welcome' Email" and you'll be able to create a welcome message.

Git

Let's commit our changes to the Git repository and push to GitHub:

```
$ git add -A
$ git commit -m "mailing list"
$ git push
```

Our tutorial application is feature complete.

Let's deploy it so we can see it running as a real website.

Chapter 22

Deploy

You've been running the default web server on your local machine. If you wanted, you could leave your computer running, set up a managed DNS service, and your web application would be accessible to anyone. But even if you wanted to leave your computer running 24 hours a day, you're probably not a security expert, your web server isn't tuned to handle much traffic, and your computer is distant from the interconnection hubs where most websites are hosted. For these reasons, when we move a web application from development to production, we deploy it to a web hosting service that provides a hosting platform on a server located in a strategically-located data center.

Data centers offer colocation services, renting rack-mounted computers with fast Internet connections that can be configured as web servers. In the early days of the web, deploying a web application required system administration skills to configure and maintain a web server. Today, some developers like to set up their web servers "from bare metal" using virtual private servers from Linode, Slicehost, Rackspace, Amazon EC2, or others. With sufficent skills and study, they say there is a feeling of satisfaction from doing it yourself. But not everyone wants to be a system administrator. Most Rails developers simply use a hosted platform as a service (PaaS) provider such as Heroku, DigitalOcean, EngineYard, OpenShift, or Pivotal Cloud Foundry.

If you've previously built web sites, you may already be using a shared web

hosting service such as GoDaddy or DreamHost. Be skeptical if a shared web hosting service, designed for WordPress or static websites, claims to support Rails applications; most do so badly. Shared hosting services offer file space for static websites on servers that are shared by thousands of websites. A Rails application requires considerably greater computing resources and specialized expertise. In contrast, a platform as a service provider offers a hardware and software stack optimized for application performance and developer convenience.

Heroku is the best known and most popular PaaS provider and we'll use it to deploy the tutorial application. Using Heroku or another PaaS provider means you don't need skills as a system administrator to manage your web server. Instead, you'll have experts maintaining the production environment, tuning system performance, and keeping the servers running.

Heroku Costs

It costs nothing to set up a Heroku account and deploy as many applications as you want. You'll pay only if you upgrade your hosting to accommodate a busy website.

Heroku pricing is based on a measure of computing resources the company calls a "dyno." Think of a dyno as a virtual server (though it is not). For personal projects, you can run your Rails application on a single dyno and never incur a charge, as long as it is not active more than 12 hours a day.

A single dyno idles after one hour of inactivity, "going to sleep" until it receives a new web request. For a personal project, this means your web application will respond with a few seconds delay if it hasn't received a web request in over an hour. After it wakes up, it will respond quickly to every browser request.

If you want your web application running 24 hours per day and responding to every request without delay, Heroku will charge \$7 per month for a "hobby" account. You'll be able to set up a custom domain, using your own domain name. Heroku offers the option of adding dynos to handle more traffic for \$25

per month; here's an article that compares Heroku costs.

A single dyno can serve thousands of requests per second, but performance depends greatly on your application. As a default, Heroku supports Puma, the recommended web server for Rails 5. Serving a typical Rails application that takes 100ms on average to process each request, Puma can accommodate about 50 requests per second per dyno, which is adequate for a personal project. If traffic surges on your website and exceeds 50 requests per second, you can scale up with more dynos.

Heroku is ideal for hosting our application:

- no system administration expertise is required

- hosting is free

- performance is excellent

For this tutorial application, we won't concern ourselves with the possibility that the website may get a lot of traffic. I'm sure you'll join me in offering hearty thanks to Heroku for providing a convenient service that beginners can use for free.

Let's deploy!

Test the Application

Before deploying an application to production, a professional Rails developer runs *integration* or *acceptance* tests. If the developer follows the discipline of *test-driven development*, he or she will have a complete test suite that confirms the application runs as expected. Often the developer uses a *continuous integration* server which automatically runs the test suite each time the code is checked into the GitHub repository.

We haven't used test-driven development to build this application so no test suite is available. You've tested the application manually at each stage.

Preparing for Heroku

You'll need to prepare your Rails application for deployment to Heroku.

Gemfile

We need to modify the Gemfile for Heroku.

We add a **group :production** block for a gem that Heroku needs:

- pg - PostgreSQL gem

Heroku doesn't support the SQLite database; the company provides a PostgreSQL database. Though we won't need it for our tutorial application, we must include the PostgreSQL gem for Heroku. We'll mark the sqlite3 gem to be used in development only.

Open your **Gemfile** and replace the contents with the following:

Gemfile

```ruby
source 'https://rubygems.org'
ruby '2.3.2'
gem 'rails', '~> 5.0.0', '>= 5.0.0.1'

# Rails defaults
gem 'puma', '~> 3.0'
gem 'sass-rails', '~> 5.0'
gem 'uglifier', '>= 1.3.0'
gem 'coffee-rails', '~> 4.2'
gem 'jquery-rails'
gem 'turbolinks', '~> 5'
gem 'jbuilder', '~> 2.5'
group :development, :test do
  gem 'byebug', platform: :mri
end
group :development do
  gem 'web-console'
  gem 'listen', '~> 3.0.5'
  gem 'spring'
```

```
  gem 'spring-watcher-listen', '~> 2.0.0'
end

# learn-rails
gem 'bootstrap-sass'
gem 'gibbon'
gem 'high_voltage'
gem 'simple_form'
group :development do
  gem 'better_errors'
  gem 'rails_layout'
  gem 'sqlite3'
end
group :production do
  gem 'pg'
end
```

We have to run **bundle install** because we've changed the Gemfile. The gem we've added is only needed in production so we don't install it on our local machine. When we deploy, Heroku will read the Gemfile and install the gem in the production environment. We'll run **bundle install** with the **--without production** argument so we don't install the new gem locally:

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

You'll see:

```
.
.
.
Gems in the group production were not installed.
Use `bundle show [gemname]` to see where a bundled gem is installed.
```

Let's commit our changes to the Git repository and push to GitHub:

```
$ git add -A
$ git commit -m "gems for Heroku"
$ git push
```

Asset Pipeline in Production

It is important to understand that assets are compiled by Heroku at the time of deployment. Rails compiles assets to make our application faster for the user.

The Rails asset pipeline produces single CSS and JavaScript files that combine all the files in the folders **app/assets/javascripts/** and **app/assets/stylesheets/**. In development mode, the Rails asset pipeline "live compiles" all CSS and JavaScript files so any changes are reflected immediately. But compiling assets adds processing overhead. In production, a web application would be slowed unnecessarily if assets were compiled for every web request. Consequently, assets are precompiled by Heroku at the time we deploy our application to production.

When assets are compiled, the Rails asset pipeline will automatically produce concatenated and minified **application.js** and **application.css** files from files listed in the manifest files **app/assets/javascripts/application.js** and **app/assets/stylesheets** The result will be several files added to the **public/assets/** folder. The filenames will contain a long unique identifier that prevents caching when you change the application CSS or JavaScript.

You can precompile assets manually with the command `rails assets:precompile` but ordinarily it is not necessary. It is a likely indicator that the asset pipeline is not working or assets are not compiled if CSS styling is missing from your web pages.

If you want to learn more about the asset pipeline, refer to the Rails Guide on the Asset Pipeline or an article from LaunchSchool.

Option to Ban Spiders

Do you want your website to show up in Google search results? If there's a link anywhere on the web to your site, within a few days (sometimes hours) the Googlebot spider will visit your site and add it to the database for the Google search engine. Most webmasters want their sites to be found in Google search

results. If that's not what you want, you may want to modify the file **public/robots.txt** to prevent indexing by search engines.

Only change this file if you want to prevent your website from appearing in search engine listings:

```
# See http://www.robotstxt.org/robotstxt.html for documentation
#
# To ban all spiders from the entire site uncomment the next two lines:
# User-agent: *
# Disallow: /
```

To block all search engine spiders, remove the commenting from the **User-Agent** and **Disallow** lines.

You can learn more about the format of the robots exclusion standard.

Humans.txt

Many websites include a **robots.txt** file for nosy bots so it's only fair that you offer a **humans.txt** file for nosy people. Few people will look for it but you can add a file **public/humans.txt** to credit and identify the creators and software behind the website. The HTML5 Boilerplate project offers an example file or you can borrow from RailsApps.

Sign Up for a Heroku Account

In the chapter, "Accounts You May Need," I suggested you sign up for a Heroku account.

To deploy an app to Heroku, you must have a Heroku account. Visit https://id.heroku.com/si to set up an account.

Be sure to use the same email address you used to configure Git locally. You can check the email address you used for Git with:

```
$ git config --get user.email
```

Heroku Toolbelt

Heroku provides a command line utility for creating and managing Heroku apps.

Visit https://toolbelt.heroku.com/ to install the Heroku Toolbelt. A one-click installer is available for Mac OS X, Windows, and Linux.

The installation process will install the Heroku command line utility. It also installs the Foreman gem which is useful for duplicating the Heroku production environment on a local machine. The installation process will also make sure Git is installed.

To make sure the Heroku command line utility is installed, try:

```
$ heroku version
heroku-toolbelt/...
```

You'll see the heroku-toolbelt version number.

You should be able to login using the email address and password you used when creating your Heroku account:

```
$ heroku login
Enter your Heroku credentials.
Email: adam@example.com
Password:
Could not find an existing public key.
Would you like to generate one? [Yn]
Generating new SSH public key.
Uploading ssh public key /Users/adam/.ssh/id_rsa.pub
```

The Heroku command line utility will create SSH keys if necessary to guarantee a secure connection to Heroku.

Heroku Create

Be sure you are in your application root directory and you've committed the tutorial application to your Git repository.

Use the Heroku create command to create and name your application.

```
$ heroku create myapp
```

Replace **myapp** with something unique. Heroku demands a unique name for every hosted application. If it is not unique, you'll see an error, "name is already taken." Chances are, "learn-rails" is already taken.

The name must start with a letter and can only contain lowercase letters, numbers, and dashes.

If you don't specify your app name (**myapp** in the example above), Heroku will supply a placeholder name. You can easily change Heroku's placeholder name to a name of your choice with the **heroku apps:rename** command (see Renaming Apps from the CLI).

Don't worry too much about getting the "perfect name" for your Heroku app. The name of your Heroku app won't matter if you plan to set up your Heroku app to use your own domain name. You'll just use the name for access to the instance of your app running on the Heroku servers; if you have a custom domain name, you'll set up DNS (*domain name service*) to point your domain name to the app running on Heroku.

The **heroku create** command sets your Heroku application as a Git remote repository. That means you'll use the **git push** command to deploy your application to Heroku.

Enable Email

You'll need to enable email for production or else you'll get errors when your application tries to send email from Heroku.

To use SendGrid, add the following to your **config/environments/production.rb** file:

```ruby
# email enabled in production
config.action_mailer.smtp_settings = {
  address: "smtp.sendgrid.net",
  port: 587,
  domain: Rails.application.secrets.domain_name,
  authentication: "plain",
  enable_starttls_auto: true,
  user_name: Rails.application.secrets.email_provider_username,
  password: Rails.application.secrets.email_provider_password
}
# ActionMailer Config
config.action_mailer.default_url_options = { :host => Rails.application.secrets.domain_name }
config.action_mailer.delivery_method = :smtp
config.action_mailer.perform_deliveries = true
config.action_mailer.raise_delivery_errors = false
```

You can use port 25, 587, or 2525 (some ISPs restrict connections on port 25).

Be sure to add the new settings before the **end** keyword in the file. The settings can be added anywhere, as long as they precede the **end** keyword!

You'll need to specify the unique name you've selected for your hosted application. We're using the **Rails.application.secrets.domain_name** configuration variable in two places in the file. The **config/secrets.yml** file provides configuration variables for use in production.

Be sure to commit your code to the Git local repository:

```
$ git add -A
$ git commit -m "email set for Heroku"
$ git push
```

Next we'll set Heroku environment variables.

Set Heroku Environment Variables

When you run your application, configuration values are obtained from the **config/secrets.yml** file, which contains Unix environment variables which are set in the **.bash_profile** or **.bashrc** files.

Heroku doesn't have a **.bash_profile** or **.bashrc** file, so you'll need a way to set environment variables on Heroku. You can use the **heroku config:add** command.

```
$ heroku config:add SENDGRID_USERNAME='example'
$ heroku config:add SENDGRID_PASSWORD='secret'
$ heroku config:add MAILCHIMP_API_KEY='my-key'
$ heroku config:add MAILCHIMP_LIST_ID='mylistid'
$ heroku config:add OWNER_EMAIL='me@example.com'
$ heroku config:add DOMAIN_NAME='myapp.herokuapp.com'
```

Don't use the values shown above. Instead, look in your **.bash_profile** or **.bashrc** files and copy the values you find there.

When you set **myapp.herokuapp.com**, replace **myapp** with the name that Heroku is using for your application. If you want to use a custom domain name, you'll need to set up DNS (*domain name service*), which we won't cover in this tutorial.

You don't need to set **SECRET_KEY_BASE**, even though it is in your **config/secrets.yml** file. Heroku sets it automatically.

Check that the environment variables are set with:

```
$ heroku config
```

See the Heroku documentation on Configuration and Config Vars and the article Rails Environment Variables for more information.

Push to Heroku

After all this preparation, you can finally push your application to Heroku.

Be sure to commit any recent changes to the Git local repository before you push to Heroku.

You commit your code to Heroku just like you push your code to GitHub.

Here's how to push to Heroku:

```
$ git push heroku master
```

You may see a message, "The authenticity of host 'heroku.com' can't be established. Are you sure you want to continue connecting (yes/no)?". You can answer "yes" and safely continue.

The push to Heroku takes several minutes. You'll see a sequence of diagnostic messages in the console, beginning with:

```
Counting objects...
```

and finishing with:

```
remote: Verifying deploy.... done.
```

Updating the Application

It is likely you'll make changes to your application after deploying to Heroku.

Each time you update your site and push the changes to GitHub, you'll also have to push the new version to Heroku. A typical update scenario looks like this:

```
$ git add -A
$ git commit -m "revised application"
$ git push
$ git push heroku master
```

Visit Your Site

Your application will be running at http://my-app-name.herokuapp.com/. You can open any web browser and visit the site. For a shortcut, you can open your default web browser and visit your site from the command line:

```
$ heroku open
```

If you're using hosted development such as Cloud9, you'll need to open a browser manually to visit the site.

If you've configured everything correctly, you should be able to sign up for the newsletter and send a contact request.

Customizing

For a real application, you'll likely want to use your own domain name for your app.

See Heroku's article about custom domains for instructions.

You may also want to improve website responsiveness by adding page caching with a content delivery network such as CloudFlare. CloudFlare can also provide an SSL connection for secure connections between the browser and server.

Heroku offers many add-on services. These are particularly noteworthy:

- Adept Scale - automated scaling of Heroku dynos

- New Relic - performance monitoring

For an in-depth look at your options, see the Rails Heroku Tutorial.

Troubleshooting

When you get errors, troubleshoot by reviewing the log files:

```
$ heroku logs
```

If necessary, use the Unix **tail** flag to monitor your log files. Open a new terminal window and enter:

```
$ heroku logs -t
```

to watch the server logs in real time.

Where to Get Help

Your best source for help with Heroku is Stack Overflow. Use the tag "heroku," "learn-ruby-on-rails," or "railsapps" when you post your question. Your issue may have been encountered and addressed by others.

You can also check the Heroku Dev Center or the Heroku Google Group.

Chapter 23

Analytics

In earlier chapters, we built the tutorial application and deployed it for hosting on Heroku.

We've left something out. Though not obvious, it's very important: analytics.

Analytics services provide reports about website traffic and usage.

You'll use the data to increase visits and improve your site. Analytics close the communication loop with your users; your website puts out a message and analytics reports show how visitors respond.

Google Analytics is the best known tracking service. It is free, easy to use, and familiar to most web developers. In this chapter we'll integrate Google Analytics with the tutorial application.

There are several ways to install Google Analytics for Rails. The article on Analytics for Rails looks at various approaches and explains how Google Analytics works.

For this tutorial, we'll use the Segment.com service. The service provides an API to send analytics data to dozens of different services, including Google Analytics.

Segment.com

Segment.com is a subscription service that gathers analytics data from your application and sends it to dozens of different services, including Google Analytics. The service is free for low-volume websites, accommodating 1000 tracked users per month at no cost. There is no charge to sign up for the service.

Using Segment.com means you install one JavaScript library and get access to reports from dozens of analytics services. You can see a list of supported services. The company offers helpful advice about which analytics tools to choose from. For low-volume sites, many of the analytics services are free, so Segment.com makes it easy to experiment and learn about the available analytics tools. The service is fast and reliable, so there's no downside to trying it.

Accounts You Will Need

You will need an account with Segment.com. Sign up for Segment.com.

You will need accounts with each of the services that you'll use via Segment.com.

You'll likely want to start with Google Analytics, so you'll need a Google Analytics account and tracking ID.

Visit the Google Analytics website to obtain the Tracking ID for your website. You'll need to know the domain name of your website to get an account for your website. If you've deployed to Heroku without a custom domain, use the domain that looks like "myapp.herokuapp.com". Or use your custom domain if you have one. Use it for fields for "Website Name," "Web Site URL," and "Account Name."

Choose the defaults when you create your Google Analytics account and click "Get Tracking ID." Your tracking ID will look like this: `UA-XXXXXX-XX`. You won't need the tracking code snippet as we will use the Segment.com

JavaScript snippet instead.

You'll check your Google Analytics account later to verify that Google is collecting data.

Installing the JavaScript Library

Segment.com provides a JavaScript snippet that sets an API token to identify your account and installs a library named **analytics.js**. This is similar to how Google Analytics works. The Segment.com library loads asynchronously, so it won't affect page load speed.

The Segment.com JavaScript snippet should be loaded on every page and it can be included as an application-wide asset using the Rails asset pipeline.

We'll add the Segment.com JavaScript snippet to a file named **app/assets/javascripts/segme** The manifest directive **//= require_tree .** in the file **app/assets/javascripts/applicatic** will ensure that the new file is included in the concatenated application JavaScript file. If you've removed the **//= require_tree .** directive, you'll have to add a directive to include the **app/assets/javascripts/segment.js** file.

Create a file **app/assets/javascripts/segment.js** and include the following:

```javascript
(function(){

  // Create a queue, but don't obliterate an existing one!
  var analytics = window.analytics = window.analytics || [];

  // If the real analytics.js is already on the page return.
  if (analytics.initialize) return;

  // If the snippet was invoked already show an error.
  if (analytics.invoked) {
    if (window.console && console.error) {
      console.error('Segment snippet included twice.');
    }
    return;
  }

  // Invoked flag, to make sure the snippet
```

```javascript
  // is never invoked twice.
  analytics.invoked = true;

  // A list of the methods in Analytics.js to stub.
  analytics.methods = [
    'trackSubmit',
    'trackClick',
    'trackLink',
    'trackForm',
    'pageview',
    'identify',
    'reset',
    'group',
    'track',
    'ready',
    'alias',
    'debug',
    'page',
    'once',
    'off',
    'on'
  ];

  // Define a factory to create stubs. These are placeholders
  // for methods in Analytics.js so that you never have to wait
  // for it to load to actually record data. The `method` is
  // stored as the first argument, so we can replay the data.
  analytics.factory = function(method){
    return function(){
      var args = Array.prototype.slice.call(arguments);
      args.unshift(method);
      analytics.push(args);
      return analytics;
    };
  };

  // For each of our methods, generate a queueing stub.
  for (var i = 0; i < analytics.methods.length; i++) {
    var key = analytics.methods[i];
    analytics[key] = analytics.factory(key);
  }

  // Define a method to load Analytics.js from our CDN,
  // and that will be sure to only ever load it once.
  analytics.load = function(key){
    // Create an async script element based on your key.
    var script = document.createElement('script');
    script.type = 'text/javascript';
    script.async = true;
    script.src = ('https:' === document.location.protocol
      ? 'https://' : 'http://')
```

```javascript
        + 'cdn.segment.com/analytics.js/v1/'
        + key + '/analytics.min.js';

    // Insert our script next to the first script element.
    var first = document.getElementsByTagName('script')[0];
    first.parentNode.insertBefore(script, first);
  };

  // Add a version to keep track of what's in the wild.
  analytics.SNIPPET_VERSION = '4.0.0';

  // Load Analytics.js with your key, which will automatically
  // load the tools you've enabled for your account. Boosh!
  analytics.load("YOUR_WRITE_KEY");

})();
```

You can get the newest version of the code from the Segment.com Quickstart page. If you copy a newer version of the code, remove the **`<script type="text/javascript">`** and **`</script>`** tags from the top and bottom. The Rails asset pipeline will add the code to the **application.js** file which already contains the necessary **`<script>`** tags.

If you copy the code from the Segment.com Quickstart instructions, also remove:

```javascript
  // Make the first page call to load the integrations. If
  // you'd like to manually name or tag the page, edit or
  // move this call however you'd like.
  analytics.page();
```

We'll add **`analytics.page();`** later, wrapping it in additional code to accommodate Rails Turbolinks.

Note that the Segment.com website offers a minified version of the snippet for faster page loads. We've used the non-minified version so you can read the code and comments. If you want, you can get minified version from the Segment.com website for improved speed.

Replace the Write Key

You **must replace** `YOUR_WRITE_KEY` with your Segment.com "Write Key."
After you log in to Segment.com, click on your "workspace," then choose a
"source," click "settings," and click "API Keys" in the side navigation bar. Add
the "Write Key" in the file where you see this line:

```
// Load Analytics.js with your key, which will automatically
// load the tools you've enabled for your account. Boosh!
analytics.load("YOUR_WRITE_KEY");
```

Now you must add extra code to the Segment.com JavaScript snippet. The
extra code accommodates Turbolinks, plus page view and event tracking, which
we'll look at next.

Add Integration Code

To make sure every page is tracked when Rails Turbolinks is used, plus track
page views and events, you must add the following JavaScript to the end of the
app/assets/javascripts/segment.js file:

```
// accommodate Turbolinks
// track page views and form submissions
$(document).on('turbolinks:load', function() {
  console.log('page loaded');
  analytics.page();
  analytics.trackForm($('#new_visitor'), 'Signed Up');
  analytics.trackForm($('#new_contact'), 'Contact Request');
})
```

Add it after the last line. Add it after the orthographic car wreck that looks like
`})();`.

I'll explain the purpose of this code next.

Page View Tracking with Turbolinks

To make sure every page is tracked when Rails Turbolinks is used, we've added the following JavaScript to the **app/assets/javascripts/segment.js** file:

```javascript
// accommodate Turbolinks
// track page views and form submissions
$(document).on('turbolinks:load', function() {
.
.
.
  analytics.page();
.
.
.
})
```

Rails 4.0 introduced a feature named Turbolinks to increase the perceived speed of a website.

Turbolinks makes an application appear faster by only updating the body and the title of a page when a link is followed. By not reloading the full page, Turbolinks reduces browser rendering time and trips to the server.

With Turbolinks, the user follows a link and sees a new page but Segment.com or Google Analytics thinks the page hasn't changed because a new page has not been loaded. To resolve the issue, you could disable Turbolinks by removing the turbolinks gem from the Gemfile. However, it's nice to have both the speed of Turbolinks and tracking data, so I'll show you how to get tracking data with Turbolinks.

Turbolinks fires a **load** event when a page has been replaced. The code listens for the **load** event and calls the Segment.com **analytics.page()** method. This code will work even on pages that are not visited through Turbolinks (for example, the first page visited).

Event Tracking

Segment.com gives us a convenient method to track page views. Page view tracking gives us data about our website traffic, showing visits to the site and information about our visitors.

It's also important to learn about a visitor's activity on the site. Site usage data helps us improve the site and determine whether we are meeting our business goals. This requires tracking events as well as page views.

The Segment.com JavaScript library gives us two methods to track events:

- **trackLink**

- **trackForm**

Link tracking can be used to send data to Segment.com whenever a visitor clicks a link. It is not useful for our tutorial application because we simply record a new page view when a visitor clicks a link on our site. However, if you add links to external sites and want to track click-throughs, you could use the **trackLink** method. The method can also be used to track clicks that don't result in a new page view, such as changing elements on a page.

The **trackForm** method is more useful for our tutorial application. We've already appended it to the **app/assets/javascripts/segment.js** file:

```javascript
// accommodate Turbolinks
// track page views and form submissions
$(document).on('turbolinks:load', function() {
  console.log('page loaded');
  analytics.page();
  analytics.trackForm($('#new_visitor'), 'Signed Up');
  analytics.trackForm($('#new_contact'), 'Contact Request');
})
```

I've included a **console.log('page loaded')** statement so you can check the browser JavaScript console to see if the code runs as expected.

The `trackForm` method takes two parameters, the ID attribute of a form and a name given to the event.

Form tracking will show us how many visitors sign up for the newsletter or submit the contact request form. Obviously we can count the number of subscribers in MailChimp or look in the site owner's inbox to see how many contact requests we've received. But form tracking helps us directly correlate the data with visitor data. For example, we can analyze our site usage data and see which traffic sources result in the most newsletter sign-ups.

You can read more about the Segment.com JavaScript library in the Segment.com documentation.

Troubleshooting

Click "Debugger" in the navigation bar so you can monitor data sent to Segment.com from your application.

When you test the application locally, you should see the results of page visits and form submissions within seconds in the Segment.com debugger.

If you don't see your page visits in the Segment.com debugger, open the browser JavaScript console, visit a page, and check for the message "page loaded" in the JavaScript console. In the Chrome browser, the JavaScript console is available under the item "Developer" in the "View" menu.

Segment.com Integrations

After installing the Segment.com JavaScript snippet in your application, go to your "sources" page and click the "integrations" link to visit the integrations page to select the services that will receive your data.

Each service requires a different configuration information. At a minimum, you'll have to provide an account identifier or API key that you obtained when

you signed up for the service.

For Google Analytics, enter your Google Analytics tracking id. It looks like `UA-XXXXXX-XX`.

With Google Analytics enabled as a Segment.com integration, you'll see form submissions appear in the Google Analytics Real-Time report, under the "Events" heading.

Note that Google doesn't process their data in real-time in most of its reports. Data appears immediately in the Google Analytics Real-Time report. Other Google Analytics reports, such as the Audience report, won't show data immediately. Check the next day for updated reports.

Deploy

Commit to the Git repo and deploy to Heroku:

```
$ git add -A
$ git commit -m "analytics"
$ git push
```

Then you can deploy to Heroku:

```
$ git push heroku master
```

When you visit the site, you should see real-time tracking of data sent to Segment.com in the Segment.com debugger.

Log into your Google Analytics account to see real-time tracking of visits to your website. Under "Standard Reports" see "Real-Time Overview." You'll see data within seconds after visiting any page.

Improving the User Experience

Website analytics can be used to improve visitors' experience of the website. Deploying the website is not the last step in your project. Unlike many earlier forms of communication (such as releasing a film, publishing a book, or broadcasting an advertisement), we can see how every visitor responds to the website. That means your work is not done when you deploy the site. Look at your usage data to see which elements of the site are getting attention and which are being used.

Does no one visit the "About" page? Maybe the navigation link is difficult to find. Do many people visit the Contact page but few submit a contact request form? Maybe you should change the label on the button or offer other ways to contact the site owner.

Effective and successful websites often are the result of systematic A/B testing (sometimes called *split testing*). A/B testing is a technique of creating variations on a web page, such as changing text, layout, or button colors, and using website analytics to measure the effect of the change. You can learn more about services such as Content Analytics in Google Analytics, Optimizely, or Visual Website Optimizer. These services provide complete "dashboards" to set up usage experiments and measure results (Optimizely is available as a Segment.com integration).

Conversion Tracking

You may only be interested in knowing that people visit your site, without measuring visitors' engagement or response to the site. But in most cases, if you build a website, you'll offer a way for visitors to respond, whether it is by purchasing a product, signing up for a newsletter, or clicking a "like" button.

The ultimate measure of website effectiveness is the conversion rate. The term comes from the direct marketing industry and originally referred to a measure of how people responded to "junk mail" offers. For a website, the conversion

rate indicates the proportion of visitors who respond to a *call to action*, which may be an offer to make a purchase, register for a membership, sign up for a newsletter, or any other activity which shows the visitor is engaged and interested.

For our tutorial application, we can measure our website effectiveness by looking at the conversion rate for newsletter sign-ups.

We're tracking page views which will give us a count of visits to the website home page. And we've got event tracking in place to count newsletter sign-ups. If 100 people visit the home page and 10 people request a newsletter, we've got a conversion rate of 10%.

We can try to improve the conversion rate by improving the user experience (perhaps through A/B testing) or focusing on increasing traffic from sources that provide a higher conversion rate.

You can monitor your site's conversion rate by setting up events as goals in Google Analytics. Segment.com also integrates with many services which provide conversion tracking.

Enjoy What You've Achieved

You've completed building the tutorial application.

If your project was to build an application for someone else, whether the company you work for, or a client like Foobar Kadigan, you've completed the deliverable.

You started with project planning, in the form of user stories. You implemented the application using a variety of technologies supported by the Ruby on Rails development platform. And you've deployed the application for others to use, with analytics in place to track traffic and usage.

Not every manager or client will appreciate the effort or the complexity of the project you've built. Mr. Kadigan's happiness may depend on how well you've

understood his goals and the degree to which you've met his expectations. If you're working for yourself, or launching your startup, you may be your own toughest boss, because there is always more to do.

With technology projects, like many other aspects of life, though it seems you'll never get it right, and never get it done, there are moments when you can savor a sense of accomplishment. This is one of those moments.

Before you start thinking about adding one more feature, or updating the application for the new releases that inevitably came out during the time you were working, take time to bask in the satisfaction of seeing the results of your work.

Software development has its own unique rhythm of frustration and satisfaction. As software developers, we subject ourselves to hours, days, or weeks of struggle with code that is cryptic and resists understanding. We gain mastery for a few minutes and then turn to the next problem. With each feature you implement, or issue you resolve, you'll experience brief elation before resuming the grind of development. But at each milestone, and at the completion of the project, you've built something tangible that you can use. You can try it out yourself and show it to others.

Give yourself full credit. You've built something extraordinary with little more than intelligence and attention. You've leveraged the work of other developers who have contributed to the open source Ruby on Rails platform and you've created your own unique product. This is what drives us as developers; to create something from nothing, using only our collective intelligence and ambition.

Chapter 24

Testing

You don't need to read this chapter if you will always be a student, or a hobbyist, working on personal projects. But if you wish to work as a professional Rails developer, or launch your own startup, with money and reputation at stake, you must learn about testing. In this chapter, I'll introduce the basic concepts of testing and show how to build a test suite for the tutorial application.

Why Test?

Software applications are fragile. When you write a song, you can include a wrong note and the song won't break. In a film, technical flaws like a "jump cut" or a microphone in the frame won't ruin an entire movie. But an unseen error in a software program can disrupt a key feature or crash the entire application.

Software applications are never finished. Songs and movies reach a stage of completion and are delivered to their audience, with no expectation that the completed work will change. With software applications, there's always an upcoming version with bug fixes or new features. As web developers, we continue to make changes to the applications that our customers are actively using.

Sometimes new features are delivered within minutes, or hours, of committing new code to the repository.

Software applications are complex. A web application, or any software program, is a machine with intricately connected parts, or *dependencies*. As an application grows, the connections quickly grow more complex, to the point where no one is able to see all the dependencies at once. Plus, web applications are often a collaborative effort, so no one person is familiar with every line of code.

Combine the evolving nature of an application, with the complexity of the product, and the likelihood that flaws will be immediately noticed by users, and you'll realize why testing is so important to the software development process.

Testing was once considered the sole responsibility of a *quality assurance* (QA) department. Senior developers created new features or fixed bugs. When the work was "done," lesser paid (and lower status) developers "in QA" clicked through screens, with written notes or scripts, as if they were users testing every feature of a program. Invariably, manual testing led to oversights, because testing notes were out of date, "edge cases" were overlooked, and the work was monotonous. In the best-run companies, QA engineers are now expert consultants on testing methods and a source of guidance for other developers. We now rely on *automated testing*. Even more important, the job of writing test code now belongs to the developer who creates a feature or fixes a bug. It's our responsibility to write adequate tests for any code we add to the repository.

What Are Tests?

Developers talk about testing as if it were an activity different from writing code. It is not. Testing is something we do while writing code. We create tests with the same text editor we use use to write code. The tests themselves are written in Ruby, just like any other part of a Rails application. You'll put the test code in either a **tests/** or **spec/** folder, committed to the Git repository with all the other code. You'll use the specialized API of a testing framework

for the methods of your tests, either Minitest or RSpec. Test code is different from code that implements features in one significant way: Instead of supporting interactions with a user, test code interacts with the code you've written, verifying the code behaves as intended.

Scripted or Exploratory

When testing is used for quality assurance, the goal is to create a suite of automated tests that will reveal any bugs that creep into code and break the application. Sometimes this is called *scripted testing*. These tests are checked into the software repository and maintained with the application. Often developers will set up a system for *continuous integration* (CI), which will automatically run the test suite whenever the repository is updated. Developers can set up a CI server such as Jenkins or use a hosted CI service such as Travis CI, CircleCI, or Semaphore to run tests automatically. Automated testing with continuous integration serves as a safety net for developers.

There is another role for testing, which is often called *exploratory testing*, or *developer testing*. These tests may end up in an application test suite, but the primary purpose is to help a developer construct an application. Many developers, after gaining experience in writing tests for quality assurance, realize that writing tests can be a useful first step in figuring out how a feature should be implemented. It may seem odd to write tests first, but exploratory testing can clarify what behavior will be required, and help the developer think through alternatives and edge cases. This approach is called *test-first development*, and many developers will tell you that when you write tests first, you'll be more satisfied; you'll be more focused; and you'll avoid tangents and detours of the "nice-to-have-but-not-really-needed" variety. We'll look closely at test-first development in conjunction with Test-Driven Development (TDD) and Behavior-Driven Development (BDD) at the end of this chapter. First, let's gain an understanding of testing terminology and practice.

Regression and Acceptance

We describe tests by the purpose they serve. In addition to exploratory testing used in test-first development, there are several kinds of tests used for quality assurance.

Regression tests are run every time we change code. Sometimes we want to make sure new features don't break the existing application. More often, we run tests after changing existing code to make it more readable, elegant, or effective. We call this tinkering "*refactoring.*" Refactoring is very similar to what we call editing or rewriting when we work with the written word. Before we refactor, we need to know what results we expect from our code, and we need automated tests to execute our code and check for the expected results. If our automated tests are adequate, we can use the tests as regression tests, making sure our refactoring hasn't introduced new bugs.

Acceptance tests are sometimes identical to regression tests, and may use the same test code. The purpose is different, so we give this kind of testing a different name. Acceptance tests provide accountability and serve a management function. These are tests that determine if a feature has been implemented as expected. It is common to run acceptance tests when an outside contractor delivers code, so we can determine if the team has delivered what we requested. We can also use acceptance tests to determine if our internal team has implemented the stated requirements. Proponents of behavior-driven development claim that the process of creating acceptance tests clarifies the product requirements. Obviously, if we want adequate acceptance tests, we need to plan carefully when specifying the product requirements. If we've planned well, we can turn our user stories into automated tests that serve as acceptance tests.

Units and Integration

We also describe tests by their relationship to the rest of the code.

Unit tests probe the internal workings of the machine. If we've written our code

well, a small section of the code, such as a class or a method, will be a discrete unit that can be tested independently of all other units. Unit tests inspect the integrity of small parts of the application in isolation. When a unit test fails, we can quickly identify and fix broken code.

We use *integration tests* to make sure the entire application works as expected. Integration tests mimic the behavior of real users. For a web application, an integration test may simulate a user signing in, filling out forms, clicking between pages, and verifying that contents of pages match expected results. Integration tests can also be called *feature tests* if they are designed to confirm that product features work as expected. Our feature tests can serve as acceptance tests if we use the test suite to determine if we've correctly implemented our user stories or other product specifications. Sometimes these tests are called *black box tests* because the code is tested as if the application was a black box, with the internal workings of the application hidden from the observer. They are also called *system tests* or *end-to-end tests*.

Sample Data

When we write tests, either feature tests or unit tests, we often want to check whether a method returns the data we expect. That means we have to create the data we need in advance of the test. Either we populate a database with the data we expect, or we disconnect the database and instantiate an object that provides the data we expect. Test frameworks give us a tool named a *factory* or a *fixture* to create sample data. Developers argue about what is better, factories or fixtures, but you'll encounter factories more often, particularly the popular FactoryGirl gem. A factory is an object that creates other objects. When you use FactoryGirl, you have the option of saving your object to the database (which is slow) or building your object in memory only (which is faster). Fixtures are used to populate a database with sample data before your tests run. If you use fixtures, you'll save sample data in a configuration file. Before tests run, Rails automatically loads all fixtures from configuration files in the **test/fixtures** folder. As you gain experience with testing, you'll become

familiar with both factories and fixtures.

Test Doubles

In unit testing, to isolate small parts of the application, sometimes we artificially decouple the code from the rest of the application. For example, with a unit test, we don't want to connect to an external service with an API to obtain data. Or we simply want a method to get a predictable response from another object.

Test doubles stand in for external dependencies. The term is borrowed from Hollywood, where stunt doubles stand in for actors in action scenes. A test double is any kind of pretend object used in place of a real object for testing purposes. There are two types of test doubles, *stubs* and *mocks*. Stubs provide canned answers to calls made during the test, only responding when queried by the test. Sometimes stubs record information about the call, for example, the message sent or the number of times called. Mocks are pre-programmed objects that reproduce the behavior of the object they represent, forming a specification of an object's behavior. It takes time to write stubs and mocks and lots of experience to use them correctly, so as a beginner, you probably won't write stubs and mocks without help. As you can gain experience, you'll better understand the difference between stubs and mocks and learn how to use them. For now, it is enough to recognize the terminology and remember that tests run faster and better when we reduce coupling and complexity with test doubles.

Minitest and RSpec

You've already learned that Rails developers mix and match gems to create a favorite technology *stack*. Not everyone likes ERB for view templates. Some prefer Haml or Slim syntax for mixing HTML and Ruby in a view. Developers often stray from the default Rails stack when it comes to testing. Since the

release of Ruby 1.9, Minitest has been supplied as a standard gem with all Ruby installations. Yet most Rails developers use RSpec for testing.

In this tutorial, I'll use Minitest to introduce you to testing. Minitest is easier to set up and offers a syntax that is very similar to RSpec. Some developers say that there is no reason to use RSpec because Minitest provides almost all the convenience of RSpec with smaller size, faster speed, and less complexity. Other developers insist that RSpec is more expressive and flexible. Realistically, if you want a job working on most Rails teams, you'll need to learn RSpec. Get started with Minitest to learn the basics of testing. When you're ready for the next step, the Capstone Rails Tutorials will take you deeper. I also recommend the books Rails 4 Test Prescriptions by Noel Rappin and Everyday Rails Testing with RSpec by Aaron Sumner.

Capybara, the Ghost in the Machine

Unit tests are simple, in principle and often in practice. The tests are just Ruby code, supplemented with methods from the test framework API. If we want unit tests for all the methods of a User class, we instantiate the class and write code that calls each method and verifies if the response matches our expectations. Using methods from the Minitest or RSpec test framework, we output a message that indicates whether each unit test passes or fails.

Integration tests, or feature tests, require more of a framework than unit tests. We want our tests to be as realistic as possible, as if a robot was using a web browser and interacting with our web application. Fortunately, the maintainers of the Capybara gem have created such a robot. To create integration tests, we add the Capybara gem, using it with either Minitest or RSpec. Capybara gives us a `visit` method that simulates a user visiting a page. After we call the `visit` method, Capybara gives us a `page` object and allows us to test whether the page contains the content we expect. Every Rails application relies on a layer of *middleware* named Rack that ties into a web server. Capybara interacts with the web application, via calls to Rack, as if it was a browser

making requests and receiving HTML files as a response.

When we use Capybara, by default it operates in *headless mode*, interacting directly with the Rails application via Rack. "Headless" means there is no graphical user interface (as if the absent screen was a computer's head). In headless mode, JavaScript is unavailable. If some of our application features require JavaScript, we must set up Capybara to act as a robot using a real web browser. Capybara has a built-in *driver* (named Selenium) that gives our robot the option of automatically launching and using a real web browser for each test. By default, Capybara will use the Firefox web browser if it is installed on your computer. What you'll see is amazing. When you run tests using Capybara with the JavaScript option, the Firefox web browser will pop open on your desktop and you'll watch a ghost flying through your web application. With Capybara, you now have a ghostly QA department running your integration tests.

Four Phases of Feature Tests

Test code is easier to understand when you recognize that tests proceed in stages, or phases. Code that simulates a user visiting a web page tends to be organized in four phases:

- set up

- visit page

- verify page contents

- neutralize

The setup phase may include creating a user, signing in, or any other activity that creates the conditions for a test. With Capybara, the test visits the page, which requires Capybara to simulate a browser request to the Rails application.

Then, in the third stage, we check if the server response contains the data we expect. Finally, we may need to clean up, resetting the original state of the application, or removing any data the test added to the database.

Four Phases of Unit Tests

Unit tests also are organized in four stages:

- set up

- exercise

- verify

- teardown

When you test a small part of the application in isolation, you'll focus on an object or method which we call the "system under test." The setup phase prepares the system under test. Often this means instantiating an object. Here is an example:

```
user = User.new(email: 'user@example.com')
```

During the exercise phase, something is executed. Often this is a method call:

```
user.save
```

During verification, the result of the exercise is verified against the developer's expectations:

```
user.email.must_equal 'user@example.com'
```

During teardown, the system under test is reset to its initial state. Rails integrates with Minitest or RSpec to reset a database to its initial state. You will seldom write code for the teardown phase.

Now that you've learned about the basic concepts of testing, let's set up Minitest for our first tests.

Set Up Minitest

We'll set up testing with both Minitest and Capybara, so we can write both unit tests and feature tests. Minitest is a standard Ruby gem, installed when you install Ruby in your environment. We'll install the minitest-spec-rails gem which makes it easy to use an RSpec-like syntax with Minitest. We'll also add the minitest-rails-capybara gem to integrate Capybara with Minitest and Rails.

Open your **Gemfile** and replace the contents with the following:

Gemfile

```
source 'https://rubygems.org'
ruby '2.3.2'
gem 'rails', '~> 5.0.0', '>= 5.0.0.1'

# Rails defaults
gem 'puma', '~> 3.0'
gem 'sass-rails', '~> 5.0'
gem 'uglifier', '>= 1.3.0'
gem 'coffee-rails', '~> 4.2'
gem 'jquery-rails'
gem 'turbolinks', '~> 5'
gem 'jbuilder', '~> 2.5'
group :development, :test do
  gem 'byebug', platform: :mri
end
group :development do
  gem 'web-console'
  gem 'listen', '~> 3.0.5'
```

```ruby
  gem 'spring'
  gem 'spring-watcher-listen', '~> 2.0.0'
end

# learn-rails
gem 'bootstrap-sass'
gem 'gibbon'
gem 'high_voltage'
gem 'simple_form'
group :development do
  gem 'better_errors'
  gem 'rails_layout'
  gem 'sqlite3'
end
group :production do
  gem 'pg'
end
group :test do
  gem 'minitest-spec-rails'
  gem 'minitest-rails-capybara'
end
```

We've added the two gems to the **test** group. Now, some gems are loaded only when we're writing code (during development), some are loaded only when the application is running on Heroku (deployed to production), and our newest additions only are loaded when we run tests.

Next, install the additional gems:

```
$ bundle install
```

The **bundle install** command will download and install the gems from the rubygems.org server.

Configure for Capybara

Our application already has a **test/** folder containing a **test_helper.rb** file. We'll modify the helper file so we can write tests that use Capybara.

Modify the **test/test_helper.rb** file:

```
ENV['RAILS_ENV'] ||= 'test'
require File.expand_path('../../config/environment', __FILE__)
require 'rails/test_help'
require 'minitest/rails/capybara'

class ActiveSupport::TestCase
  # Setup all fixtures in test/fixtures/*.yml for all tests in alphabetical order.
  fixtures :all

  # Add more helper methods to be used by all tests here...
end
```

We've added the line **require "minitest/rails/capybara"** to allow use
of the Capybara test framework methods.

Run Tests

The command **rails test** will execute Minitest. Let's see what happens
when we run tests:

```
$ rails test
Running via Spring preloader in process 29254
/Users/danielkehoe/workspace/learn-rails/db/schema.rb doesn't exist yet.
Run `rails db:migrate` to create it, then try again. If you do not intend
to use a database, you should instead alter /Users/danielkehoe/workspace/
/learn-rails/config/application.rb to limit the frameworks that will be loaded.
Run options: --seed 35136

# Running:
Finished in 0.005570s, 0.0000 runs/s, 0.0000 assertions/s.
0 runs, 0 assertions, 0 failures, 0 errors, 0 skips
```

Rails Minitest informs us that our application is not set up to use a database.
The easiest way to resolve the issue is to run **rails db:migrate** which sets
up the database schema file. We're not using a database for this application so
the database schema file will be empty.

```
$ rails db:migrate
```

The **rails db:migrate** command doesn't produce any result in the terminal but it will create a new **db/schema.rb** file.

Try running **rails test** again:

```
$ rails test
Running via Spring preloader in process 29468
Run options: --seed 46952

# Running:

Finished in 0.005945s, 0.0000 runs/s, 0.0000 assertions/s.
0 runs, 0 assertions, 0 failures, 0 errors, 0 skips
```

The output shows that Minitest executes but we have no tests for it to run.

Let's commit our changes to the Git repository and push to GitHub:

```
$ git add -A
$ git commit -m "set up minitest"
$ git push
```

Unit Test (Standard Syntax)

In its default form, Minitest uses the syntax of the older test_unit framework that was supplied with Ruby before version 1.9. The test_unit syntax uses explicit Ruby to set up tests. Here's an example of Minitest using the test_unit syntax:

```
require 'test_helper'

class VisitorTest < ActiveSupport::TestCase
```

```ruby
  def valid_params
    { email: 'john@example.com' }
  end

  def test_valid
    visitor = Visitor.new valid_params
    assert visitor.valid?, "Can't create with valid params: #{visitor.errors.messages}"
  end

  def test_invalid_without_email
    params = valid_params.clone
    params.delete :email
    visitor = Visitor.new params
    refute visitor.valid?, "Can't be valid without email"
    assert visitor.errors[:email], "Missing error when without email"
  end

end
```

Notice that we must declare a class **VisitorTest** that inherits from **ActiveSupport::Te**
Then we must define a new method for each test case using the **def** keyword.
This syntax is not popular with Rails developers. RSpec offers its own DSL
(domain specific language) that hides the overhead of setting up classes and
methods behind convenience methods. Minitest offers its own version of the
the RSpec DSL, allowing us to use the more popular syntax. I'll use the new
RSpec-like syntax in this tutorial, since you are likely to encounter RSpec more
frequently.

Unit Test (Spec Syntax)

For our first test, let's create a simple unit test for our Visitor model. Every
time we run our tests, we want to know that we're able to create a Visitor
model. We'll also check that the Visitor model contains a method that returns
an email address.

The default Rails directory structure already contains a **test/models/** folder.
Thanks to Rails conventions, we know exactly where to create our test file.

Create a file **test/models/visitor_test.rb**:

```ruby
require 'test_helper'

describe Visitor do

  let(:visitor_params) { {email: 'user@example.com'} }
  let(:visitor) { Visitor.new visitor_params }

  it 'is valid when created with valid parameters' do
    visitor.must_be :valid?
  end

  it 'is invalid without an email' do
    # Delete email before visitor let is called
    visitor_params.delete :email
    visitor.wont_be :valid? # Must not be valid without email
    visitor.errors[:email].must_be :present? # Must have error for missing email
  end

end
```

The test above, written in the RSpec-like syntax, is functionally identical to the previous example, written in the old test_unit syntax. Take a close look at both, so the structure and keywords will be familiar when you see it again.

We need `require 'test_helper'` to enable the test framework and apply any configuration settings.

The keywords **describe**, **let**, and **it** are keywords that are also used in the RSpec DSL (domain-specific language). When you see these keywords, you know you are looking at test code, either Minitest or RSpec.

The purpose of a unit test is to describe the system under test, in terms of its expected behavior. We create a **do ... end** block using the **describe** keyword and specifying a class we wish to test:

```ruby
describe Visitor do
  .
  .
  .
end
```

Create a Test Class With Describe

The **describe** keyword creates a test class. In this case, the **describe** keyword will create a class named **VisitorTest** that inherits from **ActiveSupport::TestC**
Using the old test_unit syntax, we could do this with **class VisitorTest < ActiveSupport::TestCase** but the **describe** keyword is more convenient. When Minitest runs, it recognizes and executes test classes. By including our code inside a test class, we get to use methods such as **let** and **it** which are useful for writing tests. Minitest will recognize various classes like models or controllers and provide appropriate behavior.

Setup Phase

We must set up everything we need for the test. Minitest provides a simple way to set up everything before a test using the **before** keyword:

```
before do
  do_some_setup
end
```

We could initialize the Visitor model using a **before** block and setting instance variables:

```
before do
  @visitor_params = {email: 'user@example.com'}
  @visitor = Visitor.new(visitor_params)
end
```

Instead of using a **before** block, we'll use the convenient **let** keyword:

```
let(:visitor_params) { {email: 'user@example.com'} }
let(:visitor) { Visitor.new visitor_params }
```

The **let** keyword is a specialized version of the **before** keyword. It caches the objects that you create so they are ready for every test you write in the test class. And it is *lazy-loaded*, which means it does not require any processing overhead until the first time it is used.

Do It

Each test is defined by the **it** keyword and a **do ... end** block that contains the exercise and verify phases of the test. The **it** keyword must be accompanied by a description. The description will be displayed if the test fails.

For our first test, we want to check if the Visitor model can be created when we provide a valid email address. Before the test runs, the **let** statement makes sure the Visitor object is instantiated with an email value.

The verification phase of each test consists of a comparison between the results of an operation and our expectations. We expect that each time we create a Visitor object with a valid email address, the **visitor.valid?** method will return true. We can create a test:

```
it 'is valid when created with valid parameters' do
  assert_equal visitor.valid?, true
end
```

The keyword **assert_equal** is the old test_unit syntax. It compares the result of **visitor.valid?** with **true** and tells Minitest the test has passed or failed.

We can write the same thing using the new RSpec-style syntax:

```
it 'is valid when created with valid parameters' do
  visitor.must_be :valid?
end
```

The method **must_be** is an *expectation*. You can see a Minitest cheat sheet with a list of all the expectation methods. As you might guess, **must_be** func-

tions as a comparison operator, checking if a call to **visitor.valid?** returns true.

For our second test, we want to make sure the Visitor object is invalid when no email address is provided:

```
it 'is invalid without an email' do
  # Delete email before visitor let is called
  visitor_params.delete :email
  visitor.wont_be :valid? # Must not be valid without email
  visitor.errors[:email].must_be :present? # Must have error for missing email
end
```

We created the **visitor_params** hash with a **let** statement. Before we invoke the Visitor object and call the **visitor.valid?** method, we delete the email address from the **visitor_params** hash. When the Visitor object is invoked, it will be created by the **let** statement without an email address. The **wont_be** expectation confirms that the result of **visitor.valid?** method is **false**. Then we check if a validation error message is present.

At this point, don't expect to be ready to write unit tests for every model method. You'll need to spend time with the documentation for Minitest expectations or the Minitest cheat sheet to become familiar with all the possible ways to write tests. This introduction should help you recognize the syntax of tests, understand the structure, and give you the background you need to learn more about unit testing.

Run Tests

Let's run our unit tests:

```
$ rails test
Running via Spring preloader in process 29585
Run options: --seed 7800

# Running:
```

```
..

Finished in 0.020289s, 98.5770 runs/s, 147.8655 assertions/s.

2 runs, 3 assertions, 0 failures, 0 errors, 0 skips
```

The output shows that our tests pass.

Breaking the Test

Let's see what happens if we purposefully break our Visitor model. Modify the file **app/models/visitor.rb**:

```ruby
class Visitor
  include ActiveModel::Model
  attr_accessor :email, :string
  # validates_presence_of :email
  # validates_format_of :email, with: /\A[-a-z0-9_+\.]+\@([-a-z0-9]+\.)+[a-z0-9]{2,4}\z/i

  def subscribe
    mailchimp = Gibbon::Request.new(api_key: Rails.application.secrets.mailchimp_api_key)
    list_id = Rails.application.secrets.mailchimp_list_id
    result = mailchimp.lists(list_id).members.create(
      body: {
        email_address: self.email,
        status: 'subscribed'
    })
    Rails.logger.info("Subscribed #{self.email} to MailChimp") if result
  end

end
```

When you copy this, be careful to keep the long regex expression (/\A. . .\z/i) on one line (no line breaks).

We've commented out the statements that require validation for the email attribute. Let's run the tests again:

```
$ rails test
Running via Spring preloader in process 29655
Run options: --seed 45089

# Running:

F

Failure:
Visitor#test_0002_is invalid without an email
[/Users/danielkehoe/workspace//learn-rails/test/models/visitor_test.rb:15]:
Expected #<Visitor:0x007fa5440a2260 @validation_context=nil,
@errors=#<ActiveModel::Errors:0x007fa5440a1f40 @base=#<Visitor:0x007fa5440a2260 ...>,
@messages={}, @details={}>> to not be valid?.

bin/rails test test/models/visitor_test.rb:12

.

Finished in 0.008457s, 236.4993 runs/s, 236.4993 assertions/s.

2 runs, 2 assertions, 1 failures, 0 errors, 0 skips
```

The output shows a failure. The diagnostic message displays the description of the failing test, "Visitor#test_0002_is invalid without an email", and indicates the line number where the test failed. Now you know what a failing test looks like.

Before you continue, restore the file **app/models/visitor.rb** to its original state, and make sure the tests pass.

If you wish, you can continue writing unit tests. You could create a similar unit test for the Contact model. With more experience, or some independent research, you could create a test for the **subscribe** method in the Visitor model. This method connects to an external API, so it requires test doubles to fake the response of the external services. Our goal here is to introduce you to the concepts of testing, so we'll put aside advanced work on unit tests, and take a look at feature tests.

Feature Test

Let's start with a user story for our home page. It might seem trivial to call the home page a "feature" and describe it with a user story, but it illustrates a process that works just as well with more complex features. Here's our user story:

```
*Feature: Home page*
  As a visitor
  I want to visit a home page
  And see a welcome message
```

For our test, we know we want to visit the home page and check if the words "Stay in touch" appear on the page. This is the scenario we'll test:

```
*Scenario: Visit the home page*
  Given I am a visitor
  When I visit the home page
  Then I see "Stay in touch"
```

If you think of your application as a collection of features, and you describe each feature in terms of "As a (role), I want (goal), In order to (benefit)," and then imagine scenarios for each feature using the "Given..., When..., Then..." formula, you'll be able to write automated tests to cover every feature in the application. Let's try it for the home page.

Examine the folders within the **test/** directory. Remember that feature tests are also called integration tests. You'll see a folder **test/integration/**. That's where we'll add our feature tests.

Create a file **test/integration/home_page_test.rb**:

```
require 'test_helper'

# Feature: Home page
#   As a visitor
```

```ruby
#   I want to visit a home page
#   So I can learn more about the website
feature 'Home page' do

  # Scenario: Visit the home page
  #   Given I am a visitor
  #   When I visit the home page
  #   Then I see "Welcome"
  scenario 'visit the home page' do
    visit root_path
    page.must_have_content 'Stay in touch'
  end
end
```

I've included the user story and scenario description in comments. There's no convention to do so, but it will help you to see the relationship between testing and the product planning process. It should be easy to transform a "Given...
When... Then..." scenario into the code needed for a feature test.

Feature

When we created a unit test, we used the **describe** keyword to create a test class. The **feature** keyword creates a test class that inherits from the **Capybara::Rails::TestCase** class, giving us methods such as **visit** and **page**.

Feature tests are created with a **do ... end** block using the **feature** keyword and providing a description of the feature:

```ruby
feature 'Home page' do
  .
  .
  .
end
```

Notice that the description is placed in quotes. In this case, Minitest will automatically generate a class named **HomePageTest**.

Scenario

Typically we test a single feature with multiple scenarios in a single test file.

The **scenario** keyword is similar to the **it** keyword you've seen in unit tests. Each feature test is defined by the **scenario** keyword and a **do ... end** block that contains the visit and verify phases of the test. The **scenario** keyword must be accompanied by a description. The description will be displayed if the test fails.

```
scenario 'visit the home page' do
  visit root_path
  page.must_have_content 'Stay in touch'
end
```

The directive **visit** is a Capybara method that takes a URL or Rails route as an argument. You could specify either **visit '/'** or **visit root_path** to direct Capybara to retrieve the home page.

Capybara provides other *actions* in addition to **visit**. You can see the [documentation for Capybara actions](#) that include actions for filling in a form and clicking a button.

Capybara creates a **page** object for us as a response to the visit. The **page** object is a representation of the HTML file returned by the application. We can call the **must_have_content** method, testing if the string "Stay in touch" is present in the page.

Capybara gives us a collection of methods we can use to verify our expectations. The [documentation for Capybara expectations](#) provides an extensive collection of methods we can use to verify what's on a web page. For example, **must_have_link** checks for a link. With Capybara expectations, you can check almost anything on a page. Combining Capybara actions and expectations allows you to build a powerful page-checking robot.

Run Tests

Let's run all our tests:

```
$ rails test
Running via Spring preloader in process 30144
Run options: --seed 51858

# Running:

...

Finished in 0.459255s, 6.5323 runs/s, 8.7098 assertions/s.

3 runs, 4 assertions, 0 failures, 0 errors, 0 skips
```

We have three tests (in two test files) making four assertions, all passing.

Troubleshooting

You might get an error message:

```
rails aborted!
NoMethodError: undefined method `feature' for main:Object
```

You'll see this error message if you neglected to modify the **test/test_helper.rb** file to allow use of the Capybara test framework methods.

Breaking the Test

Let's see what happens if we purposefully break our home page. Modify the file **app/view/visitors/new.html.erb**:

```
<% content_for :title do %>Foobar Kadigan<% end %>
<% content_for :description do %>Website of Foobar Kadigan<% end %>
<section>
  <img src="http://lorempixel.com/1170/600/cats/1">
</section>
<section>
  <div class="column">
    <h3>GO AWAY!</h3>
  </div>
  <div class="column">
    <div class="form-centered">
      <%= simple_form_for @visitor do |f| %>
        <%= f.error_notification %>
        <%= f.input :email, label: false, :placeholder => 'Your email address...' %>
        <br/>
        <%= f.button :submit, "Sign up for the newsletter", :class => "submit" %>
      <% end %>
    </div>
  </div>
</section>
```

We've changed the welcome message from "Stay in touch" to "GO AWAY!".

Let's run the tests again:

```
$ rails test
Running via Spring preloader in process 30059
Run options: --seed 39958

# Running:

F

Failure:
Home page Feature Test#test_0001_visit the home page
[/Users/danielkehoe/workspace//learn-rails/test/integration/home_page_test.rb:15]:
Expected to find text "Stay in touch" in "Toggle navigation Home About Contact GO AWAY!".

bin/rails test test/integration/home_page_test.rb:13


..

Finished in 5.124522s, 0.5854 runs/s, 0.7806 assertions/s.

3 runs, 4 assertions, 1 failures, 0 errors, 0 skips
```

The output shows a failure. The diagnostic message displays the description

of the failing test, "Home page Feature Test#test_0001_visit the home page", showing a failure, "Expected to include 'Stay in touch'."

Before you continue, restore the file **app/view/visitors/new.html.erb** to its original state, and make sure the tests pass.

Now that we have written a few basic tests, let's commit our changes to the Git repository and push to GitHub:

```
$ git add -A
$ git commit -m "add tests"
$ git push
```

You've written a complete application with tests. Very good!

Using Capybara

There is an art to developing feature tests. You can test that all the text on the home page is exactly what you want. That would make your test files large. And your tests would be "brittle," because any changes you made in development, even the slightest changes to the words on the page, would break your tests. For good integration tests, focus on the features that are essential to your application. For example, use the Capybara robot to make sure the user can follow a critical path through your application, visiting important pages, filling in forms, clicking buttons, and seeing results. Capybara lets you select any HTML element on a page, so you can check an ID or class attribute of an HTML tag, not just text on a page. You'll want to be confident that application navigation and page flow continues to work after any code changes. That will serve you better than tests that tell you a word changed here or there.

Other Tests

The art of testing lies in making good choices about what to test. It's common to write feature tests because they will test the entire application from the viewpoint of the user. It is also common to write unit tests for models because models contain much of the uniqueness of an application.

Every other aspect of a Rails application can be tested, including controllers, helpers, and views. Developers seldom write tests for every aspect of a Rails application. If your controllers contain only the standard RESTful actions, with no extra logic, you probably don't need to write unit tests for your controllers. If you only have simple HTML markup in helpers, helpers don't need to be tested. And views are rarely tested with unit tests (use feature tests if you want to make sure a page contains what you expect). As a beginner, you'll make a good start if you concentrate on unit tests for models and integration tests for your page flow.

Behavior-Driven Development

In Book One, you learned about the software development approach called Behavior-Driven Development (BDD), or sometimes, Behavior-Driven Design. In writing the feature tests for the home page, you saw it in action. With BDD, you turn user stories into detailed scenarios that are accompanied by tests. BDD is a powerful approach for managing software development. It helps you define your product requirements, refine your project scope, and divide your project into well-defined tasks. The BDD process is complete when each feature has automated tests, when you enter **rails test** on the command line and see that every feature is implemented and functioning as expected.

You may feel lost or overwhelmed when you attempt to build a Rails application for the first time, especially if your only experience is following the step-by-step instructions of a tutorial. When you experience that panic, BDD is your lifeline. Start by writing user stories for a few simple features. Write feature

tests and implement the code required to make the tests pass. As you focus on the process of writing scenarios and tests, and implementing the code for each feature, you'll begin to gain momentum, and before you know it, you'll be over the first hurdle.

Test-Driven Development

You can see how the BDD approach refines the product requirements and user experience. At a microscopic level, a similar discipline, named *test-driven development*, helps refine the implementation. Where BDD is driven by feature tests, TDD is focused on unit tests.

TDD is an approach to software development that emphasizes that all code should be written to be tested. Excellent test coverage, allowing easy refactoring, is not the only goal of TDD. Just as important, the developer focuses on what needs to be accomplished and thinks through alternatives and edge cases. Some TDD aficionados say testing is a tool to write better code, and regression tests are a side effect. Unit tests are at the heart of TDD, and easiest to write when code is carefully decoupled into systems that can be tested in isolation. An application that is composed of decoupled units with clearly defined interfaces is a well-designed application that is easy to extend and maintain. If you make it a practice to write unit tests in conjunction with all the code you write, you'll write better code, and you'll be practicing TDD.

Test-First Development

Often when you are practicing TDD, you'll write tests before you write implementation code. Earlier in this chapter, I referred to *test-first development* and explained that it serves a different purpose than testing for quality assurance. In some situations, test-first development is simply exploratory testing, a means of describing the behavior of the code that must be built. Test-first develop-

ment is particularly useful when you've solved a similar problem and know exactly what results to expect, making it easy to write tests before writing the implementation.

Test-first development leads to a "red-green-refactor" workflow. A developer imagines the results of an operation, writes a test that checks for the results, and runs tests which fail (with the right configuration, failing tests display as red in the console). Then the developer writes code that produces the correct results and runs the tests again, improving the code until the tests pass (displaying in green). At this point, the developer has an adequate regression test and can begin to refactor to improve the implementation, checking that the tests continue to pass. Developers like the rhythm and coherency of the "red-green-refactor" workflow. Writing tests creates discrete, manageable tasks. When tests pass, turning green, there is a feeling of satisfaction, of a job well-done. By postponing concerns about improving the code to a refactoring phase, it's easier to get the job done without trying to get it perfect. And perfection can be pursued in the refactoring phase without worrying about regressing to a broken state.

David Heinemeier Hansson, the creator of Rails, famously declared that TDD is dead. Long live testing.. He pointed out that sometimes ardent advocates of TDD will try out an implementation before writing tests, to determine what needs to be done, or to clarify a problem. In the real world, even though developers recommend writing tests first, there are often times when a developer will write tests only after writing code and settling on an approach. TDD, which emphasizes the benefit of writing tests as a first step, doesn't really require that you write tests before you write code, or even that you write tests for all code. The test-first emphasis of TDD is a recommendation, not a rule. You'll be a better developer if you find opportunities to get "in the zone" with the red-green-refactor workflow of test-first development, but testing is worthwhile whether it comes first or last.

Words of Encouragement

Testing often intimidates the newcomer. It is difficult to find good examples. The syntax of Minitest and RSpec has evolved over time, so there is little consistency among examples you'll find. Older examples are not a good guide to current practices. But once you gain familiarity with the concepts in this chapter, you can start writing tests.

Testing is one of the few things in Rails that you can jump into without getting just right. You can't screw up your code base by writing incorrect tests. Experienced developers seem to worry that inexperienced developers will write slow tests, but in truth, a slow test is better than no test. Tests won't affect the performance of your application in production.

If your code is clumsy, don't worry, you'll get better with practice. What's most important is that you've begun writing tests. That's an indication you are committed to Rails best practices.

Your tests are only "bad" if they don't cover your code adequately or if they give you a false sense of assurance. You will only discover this over time, as you find bugs you didn't anticipate (which is inevitable). It's better to just begin testing, even if you're not sure you're doing it right, than to not test at all.

Chapter 25

Rails Composer

I'm going to show you how to skip all the work you already did, and build a ready-to-use Rails application in less than five minutes. When you're done with this chapter, you may wonder why you read the rest of the book.

Watch the video for a demonstration of Rails Composer:

- Rails Composer

Building Starter Applications

This chapter is about Rails Composer, a tool for building starter applications. Rails Composer makes building applications so easy, it feels like a guilty pleasure.

In the introductory "Create the Application" chapter, you learned that developers often use a starter application instead of assembling an application from scratch. You've seen how the **rails new** command gives you a rudimentary starter application. Developers typically add a front-end framework, a testing framework, and a handful of favorite gems before they get started on any custom development. Since most applications start with the same basic com-

ponents, it makes sense to rely on an open source effort to stitch them together, so any integration issues or update problems are quickly resolved by the community. That's the idea behind the RailsApps project. The project provides a collection of starter applications, plus Rails Composer, a tool that creates the starter applications.

I've been leading the RailsApps project for several years because I think the project saves time for developers and makes things easier. I may be biased, so take a look and judge for yourself.

Build 'Learn Rails' in Less Than Five Minutes

In less than five minutes, we can build our tutorial application using Rails Composer. It will be identical to the application you've built, but we'll call it "foobar-kadigan." It's a new application, so if you're still in the **workspace/learn-rails/** project directory, move up a level to the **workspace/** project directory:

```
$ cd ../
$ pwd
/Users/danielkehoe/workspace
```

Or jump to it directly, if it's one level below your home directory:

```
$ cd ~/workspace
$ pwd
/Users/danielkehoe/workspace
```

Use the "learn-rails" gemset we created earlier:

```
$ rvm use ruby-2.3.2@learn-rails
```

Now create the "foobar-kadigan" application:

```
$ rails new foobar-kadigan -m https://raw.github.com/RailsApps/rails-composer/master/composer.r
```

We're using the **rails new** command and designating "foobar-kadigan" as the name for the application. The **-m** flag applies an *application template*, which is a script that generates an application. The application template can be on your local computer, or retrieved from a remote server. Rails Composer is an application template that is stored on GitHub. When you run the **rails new** command as shown above, a new Rails application is built and then modified by the Rails Composer script.

Here's the first prompt you'll see:

```
option  Build a starter application?
    1)  Build a RailsApps example application
    2)  Contributed applications
    3)  Custom application
choose  Enter your selection:
```

Options #2 and #3 are not for beginners. We'll skip any contributed applications. And the "Custom application" option is strictly for experts. Enter **1** to select "Build a RailsApps example application." You'll see a list of available starter applications:

```
option  Choose a starter application.
    1)   learn-rails
    2)   rails-bootstrap
    3)   rails-foundation
    4)   rails-mailinglist-activejob
    5)   rails-omniauth
    6)   rails-devise
    7)   rails-devise-roles
    8)   rails-devise-pundit
    9)   rails-signup-download
   10)   rails-stripe-checkout
   11)   rails-stripe-coupons
choose  Enter your selection:
```

We'll explore the list later. For now, enter **1** to select "learn-rails".

Here's your chance to get news and announcements about Rails Composer:

```
        Get on the mailing list for Rails Composer news?
option  Enter your email address:
```

Enter your email address (if you want news) or press "return" to skip it (if you're shy).

In less than the time it took me to write a few sentences, you'll have a new Rails application. Look for it in your folder:

```
$ ls -1
foobar-kadigan
learn-rails
```

You've just created a new application named "foobar-kadigan" that is almost identical to the "learn-rails" application you created from scratch. If you have a file compare tool on your computer, you can compare the folders and see that the only differences are the application name embedded in the application, plus a few configuration settings such as the secret keys in the **config/secrets.yml** file.

Move into the project directory:

```
$ cd foobar-kadigan
```

As soon as you move into the **foobar-kadigan/** folder, RVM will automatically begin using the gemset named "foobar-kadigan." That's because Rails Composer created hidden **.ruby-gemset** and **.ruby-version** files.

Run **bundle install** to install the necessary gems in the RVM gemset:

```
$ bundle install
```

Try running the application.

```
$ cd foobar-kadigan
$ rails server
=> Booting Puma
.
.
.
```

Open a web browser window and navigate to http://localhost:3000/. Try it out. You'll see our new home page with the placeholder photo and the "sign up" form.

The application will be almost identical to the one you already built. Compare the project files side-by-side in your editor. The files will be nearly identical. In fact, if you made mistakes when you built the tutorial application, Rails Composer will give you the newest and most correct version of the application so you can check for your mistakes with a file compare tool.

You are probably already aware that a perfect version of the tutorial application is already on GitHub, in the learn-rails GitHub repository. You could use **git clone** to get a copy to use as a starter application. The version generated by Rails Composer differs in one important respect. Rails Composer generates the application with any name you give it, so there's no need to search and change every use of the name in the application.

I hope you're not irritated that I asked you to spend hours building the "learn-rails" application, and then showed you how to build the same application in less than five minutes. I promise you the time you spent with the book is worthwhile, because you've gained a knowledge of Rails you can't get from using Rails Composer.

A Collection of Starter Applications

Since you've already built the "learn-rails" application, the identical "foobar-kadigan" application may not be interesting. Let's look at the other applications you can generate with Rails Composer.

Rails Bootstrap

The "rails-bootstrap" application provides an integration of Rails and Bootstrap, the popular front-end framework. This application gives you everything you built in this book's chapters on "Layout and Views" and "Front-End Framework," including flash messages and navigation, set up for Bootstrap. It's a good beginning point for many custom applications.

You can examine the example application on GitHub, in the rails-bootstrap repository.

If you have subscribed for the Capstone Rails Tutorials, you can read the Bootstrap Quickstart Guide to understand the code.

Rails Foundation

The "rails-foundation" application is just like the "rails-bootstrap" application, only with the Zurb Foundation front-end framework instead of Bootstrap. It's a stripped-down version of the "learn-rails" application you just built, without the contact form or mailing list sign-up, using Foundation. If you want to build a custom application, starting with nothing more than Foundation and an "about" page, generate the "rails-foundation" application.

You can examine the example application on GitHub, in the rails-foundation repository.

There's a Foundation Quickstart Guide with the Capstone Rails Tutorials.

Rails Mailing List with Active Job

Rails 4.2 added the Active Job feature for background processing. The Mailing List with Active Job tutorial explains how to use it. You can use Rails Composer to generate the rails-mailinglist-activejob starter application.

In the "Send Mail" chapter I wrote about "Asynchronous Mailing" which provides a queueing system for background jobs. For a production website, it is smart to use Active Job for better website performance for users. The Mailing List with Active Job tutorial in the Capstone Rails Tutorials series explains the code.

Rails OmniAuth

OmniAuth is a gem for authentication. Most web applications need a way for users to sign in, allowing access to some features of the application only for signed-in users. OmniAuth allows a user to sign in using an account they already have with a service such as Facebook, Twitter, or GitHub. If you're building an application that needs quick and easy sign-in, this is a useful starter application.

You can examine the example application on GitHub, in the rails-omniauth repository.

You can read the OmniAuth Tutorial in the Capstone Rails Tutorials series to learn about authentication with OmniAuth.

Rails Devise

Devise is the most popular gem for authentication. Devise provides user management and authentication, letting a user sign up to create an account and log in with an email address and password. Most websites need email/password authentication, so this is a popular starter application.

You can examine the example application on GitHub, in the rails-devise repository.

You can read the Devise Quickstart Guide in the Capstone Rails Tutorials series to learn about user management and authentication with Devise.

Rails Devise Roles

Devise is a popular gem for *authentication*, verifying a user's registered identity. In conjunction with authentication, *authorization* limits access to pages of a web application. With role-based authorization, a user can be assigned a role such as "user," "admin," or "VIP" (a "very important person"). If you want to control access to features of the website by checking a user's role, this is a useful starter application.

You can examine the example application on GitHub, in the rails-devise-roles repository.

You can read the Rails Authorization Tutorial in the Capstone Rails Tutorials series to learn about authorization.

Rails Devise Pundit

To keep controllers skinny, Rails developers often use the Pundit gem for authorization. It improves upon simple role-based authorization to move access control code from controllers to separate "policy objects." For complex applications with elegant architecture, use the "rails-devise-pundit" starter application.

You can examine the example application on GitHub, in the rails-devise-pundit repository.

You can read the Pundit Quickstart Guide in the Capstone Rails Tutorials series to learn about authorization with Pundit.

Other Starter Applications

Other applications in the Rails Composer collection include:

- rails-signup-download

- rails-stripe-checkout

- rails-stripe-coupons

You can use the rails-signup-download application to build a website where a user can download a PDF file after registering with an email address. Using the code in the Signup and Download Tutorial, you could customize the "learn-rails" application so visitors could download an ebook by Foobar Kadigan after they sign up for his newsletter.

Stripe is a popular service used to accept credit card payments. Stripe offers two approaches to implementing payment processing. Stripe Checkout is Stripe's entry-level approach. Stripe Checkout competes with the button-based payment options from Google, PayPal, or Amazon, adding a pop-up payment form to any web page. Stripe Checkout is very limited because the pop-up payment form cannot be customized for use with a Rails application. Our Stripe Checkout Tutorial in the Capstone Rails Tutorials series shows how to combine Stripe Checkout with Devise for simple applications.

Stripe.js is optimal for use with a Rails application, allowing full customization of a payment form and integration with Rails form processing. The rails-stripe-coupons application implements a payment feature using Stripe JS so a visitor pays to download a PDF file. The application accommodates promotional coupons and adds payment forms to landing pages, for real-world payment processing. Our Stripe JS With Coupons tutorial in the Capstone Rails Tutorials series provides the details.

Rails Composer Options

If all Rails Composer did was copy example applications from GitHub repos, it would be convenient but not very interesting. When you built the "foobar-kadigan" application with Rails Composer, it simply built a replica of our tutorial application. When you build the other starter application, the options get more interesting. Rails Composer lets developers customize their starter applications for their favorite stack (we discussed stacks in the "Concepts" chapter in Book One).

Let's see what options we get when we build the powerful rails-devise-roles starter application.

Jump to your **workspace/** folder so we can create a new application:

```
$ cd ~/workspace
$ pwd
/Users/danielkehoe/workspace
```

It's okay to start with the "learn-rails" gemset. We have to start with a gemset that already has the Rails gem installed. After that, Rails Composer will create a new gemset for the new project.

```
$ rvm use ruby-2.3.2@learn-rails
```

Now generate the "rails-devise-roles" starter application:

```
$ rails new rails-devise-roles -m https://raw.github.com/RailsApps/rails-composer/master/compos
```

Don't worry if some of the prompts are different from the ones I describe here. Rails Composer changes often. At the time I wrote this, I saw:

```
option  Build a starter application?
    1)  Build a RailsApps example application
    2)  Contributed applications
    3)  Custom application
choose  Enter your selection:
```

Enter **1** to select "Build a RailsApps example application."

```
option  Choose a starter application.
     1)  learn-rails
     2)  rails-bootstrap
     3)  rails-foundation
     4)  rails-mailinglist-activejob
     5)  rails-omniauth
     6)  rails-devise
     7)  rails-devise-roles
     8)  rails-devise-pundit
     9)  rails-signup-download
    10)  rails-stripe-checkout
    11)  rails-stripe-coupons
choose  Enter your selection:
```

Select "rails-devise-roles" (it was #7 when I wrote this, but the list may have changed).

```
        Get on the mailing list for Rails Composer news?
option  Enter your email address:
```

Another chance to get on the mailing list. Just hit "return" if you already signed up.

```
option  Web server for development?
    1)  Puma (default)
    2)  Thin
    3)  Unicorn
    4)  Phusion Passenger (Apache/Nginx)
    5)  Phusion Passenger (Standalone)
choose  Enter your selection:
```

Our first option! We've always used Puma since it is the Rails default. Choose "Puma" to keep things familiar.

```
option  Web server for production?
    1)  Same as development
    2)  Thin
    3)  Unicorn
    4)  Puma
    5)  Phusion Passenger (Apache/Nginx)
    6)  Phusion Passenger (Standalone)
choose  Enter your selection:
```

We could get fancy for deployment. Choose "Same as development" to stay in our comfort zone.

```
option  Database used in development?
    1)  SQLite
    2)  PostgreSQL
    3)  MySQL
choose  Enter your selection:
```

We haven't explored applications that use databases in this book, but Devise and role-based authorization require saving a User model to a database. Choose "SQLite," which is built-in and ready to run in the Mac or Ubuntu environments. Developers prefer PostgreSQL for production applications, but it takes extra effort to set up, so we'll stick with SQLite for now.

```
option  Template engine?
    1)  ERB
    2)  Haml
    3)  Slim
choose  Enter your selection:
```

In this book, all our view templates were written using the ERB template language. In the "Concepts" chapter in Book One, you learned that components of Rails can be mixed for different stacks. Some developers substitute Haml or Slim for ERB. I've written an article on Haml and Rails if you'd like to know more. Choose "ERB" for now.

```
option  Test framework?
    1)  None
    2)  RSpec with Capybara
choose  Enter your selection:
```

You've had an introduction to testing with Minitest in the "Testing" chapter of this book. RSpec is popular among many developers, so Rails Composer offers an "RSpec with Capybara" option. Rails Composer will install a test suite for the rails-devise-roles application when RSpec is selected. You can read the RSpec Quickstart Guide in the Capstone Rails Tutorials series to get started with RSpec. Choose "none" for now.

```
option  Front-end framework?
    1)  None
    2)  Bootstrap 4.0
    3)  Bootstrap 3.3
    4)  Bootstrap 2.3
    5)  Zurb Foundation 5.5
    6)  Zurb Foundation 4.0
    7)  Simple CSS
choose  Enter your selection:
```

You learned to use Bootstrap 3 in this book. Let's stick with that. Choose "Bootstrap 3.3."

```
option  Add support for sending email?
    1)  None
    2)  Gmail
    3)  SMTP
    4)  SendGrid
    5)  Mandrill
choose  Enter your selection:
```

Devise will need to send email for its "forgot password" feature. Configuring email took some time for our tutorial application. Rails Composer will instantly set up everything we need to send email using our choice of services. Choose "SendGrid" for now.

```
option  Devise modules?
    1)  Devise with default modules
    2)  Devise with Confirmable module
    3)  Devise with Confirmable and Invitable modules
choose  Enter your selection:
```

Choose "Devise with default modules." Devise has options, like a Confirmable module that requires users to click a link in an email message to confirm a new account. The Invitable module provides a feature that allows administrators or other users to invite users to establish accounts. We won't need these extra features.

```
option  Admin interface for database?
    1)  None
    2)  Thoughtbot Administrate
choose  Enter your selection:
```

Thoughtbot Administrate adds an administrative interface to a database application. Choose "None" for now.

```
option  Use a form builder gem?
    1)  None
    2)  SimpleForm
choose  Enter your selection:
```

In this book, we used the SimpleForm gem to make it easy to build forms. Let's add it to the starter application by selecting "SimpleForm."

Next you'll see a menu of page layout options from Rails Composer. These are available if you use Bootstrap 3.

```
option  Add Bootstrap page templates?
    1)  None
    2)  1 Col Portfolio
    3)  2 Col Portfolio
    4)  3 Col Portfolio
```

```
    5)   4 Col Portfolio
    6)   Bare
    7)   Blog Home
    8)   Business Casual
    9)   Business Frontpage
   10)   Clean Blog
   11)   Full Width Pics
   12)   Heroic Features
   13)   Landing Page
   14)   Modern Business
   15)   One Page Wonder
   16)   Portfolio Item
   17)   Round About
   18)   Shop Homepage
   19)   Shop Item
   20)   Simple Sidebar
   21)   Small Business
   22)   Stylish Portfolio
   23)   The Big Picture
   24)   Thumbnail Gallery
choose  Enter your selection:
```

Youll get an option to install any of 23 different Bootstrap page templates. Some of these are simple one page layouts. Others are complex, multipage websites. If youd like to see what all the templates look like, browse the Start Bootstrap website to see the gallery of Bootstrap themes & templates.

I like the "Modern Business" template for several ready-made website pages, so choose 14 to install the template.

```
option  Install page-view analytics?
    1)   None
    2)   Google Analytics
    3)   Segment.com
choose  Enter your selection:
```

In our "Analytics" chapter, I said every application needs a way to analyze traffic. Let's choose "Segment.com" since we learned about it already.

```
     option  Segment.com Write Key?
```

You can enter your Segment.com Write Key here, if you know it. Otherwise, hit return and you'll get a placeholder you can replace later.

```
option  Prepare for deployment?
    1)  no
    2)  Heroku
    3)  Capistrano
choose  Enter your selection:
```

This option sets up your starter application for deployment to Heroku. Choose "no" for now.

Some developers find Rails Turbolinks annoying when they wish to integrate JavaScript with their Rails applications.

```
     option  Disable Rails Turbolinks? (y/n)
```

We're not adding any JavaScript so choose "n" to use Rails Turbolinks.

```
     option  Set a robots.txt file to ban spiders? (y/n)
```

In the "Deploy" chapter you learned that you can leave your website out of Google search results with the **robots.txt** file. Let's answer "y" or "yes" and play it safe.

```
     option  Create a GitHub repository? (y/n)
```

Rails Composer will create a GitHub repository for your starter application if your credentials are set up correctly. Let's play it safe and answer "n" or "no" to skip the repository option.

Rails Composer has all the answers it needs. On my computer, with a fast Internet connection in the heart of San Francisco, Rails composer takes about thirty seconds to build the starter application. It installs every needed gem; sets configuration files; and generates views, models, controllers, and routes. The developers who maintain the Rails Composer project have worked out any tricky integration issues so you can expect the starter application to work without any problems.

Try It Out

You've added a new application to your collection of projects:

```
$ ls -1
foobar-kadigan
learn-rails
rails-devise-roles
```

Let's examine the application.

```
$ cd rails-devise-roles
$ git log --oneline
087167a rails_apps_composer: extras
6d4f710 rails_apps_composer: add Bootstrap page layouts
be68589 rails_apps_composer: navigation links
3126403 rails_apps_composer: set up database
53a029f rails_apps_composer: add README files
7c78031 rails_apps_composer: add analytics
99e9c0c rails_apps_composer: add pages
1886d2c rails_apps_composer: front-end framework
38242d7 rails_apps_composer: add roles to a User model
839749f rails_apps_composer: devise
62753db rails_apps_composer: set email accounts
0754893 rails_apps_composer: generators
09104b9 rails_apps_composer: Gemfile
b61cd18 rails_apps_composer: initial commit
```

Rails Composer set up a Git repository and committed files as it built the application. We can see a list of Git commits with the **git log --oneline**

command.

When you move into the **rails-devise-roles/** folder, RVM will automatically begin using the gemset named "rails-devise-roles" because of the hidden **.ruby-gemset** and **.ruby-version** files.

Run **bundle install** to install the necessary gems in the RVM gemset:

```
$ bundle install
```

Let's try running the application:

```
$ rails server
=> Booting Puma
.
.
.
```

Open a web browser window and navigate to http://localhost:3000/. You'll see a navigation bar with "Sign in" and "Sign up" links that implement an authentication feature using Devise.

Sign in with the email address "user@example.com" and the password "changeme". You'll see a link to the Users page in the navigation bar that is only seen by administrators. Click the "Users" link and you'll see a list of users (just one initially). The first user (created by Rails Composer) is automatically assigned administrator privileges.

Sign out and sign up to create a new account with your own email address and password. You'll see a message "Welcome! You have signed up successfully." You will not see the "Users" link. Your new account is an ordinary user without administrator privileges, so you are not allowed to see the list of all users. Notice the navigation link "Edit account." It displays a page for account management where you can change your name, email address, or password.

Sign out and sign in again with the administrative account "user@example.com" and the password "changeme". Now you can view the list of users. You can change the role of any user.

You've got a useful starter application. Without Rails Composer, an experienced developer needs at least an hour or two to set up a similar starter application (and possibly more time if version updates have created integration issues).

Examine the application in your editor. Here's where a starter application can be useful as a learning tool. Given what you've learned so far, what do recognize as familiar? Every Rails application shares a similar structure, so you will recognize files such as the Gemfile; and folders such as **app/models/**, **app/controllers/**, and **app/views/**. Explore the application. Try to guess the purpose of the unfamiliar files and code.

If you're overwhelmed by unfamiliar files and code, try building one of the simpler starter applications, such as rails-bootstrap or rails-devise. Every line of code is explained in the Capstone Rails Tutorials series so there's no mystery code.

As a beginner, you can use Rails Composer for two purposes. You can quickly build apps that are guaranteed to work and then pick them apart. A "breakable toy" can be a wonderful instrument for learning. Make an effort to understand everything in the RailsApps example applications and you'll gain a solid understanding of the basic components used in real-world Rails projects. Secondly, start building custom applications based on the Rails Composer starter applications. By starting with Rails Composer, you'll skip the frustrating preliminaries of setting up a front-end framework, authentication, or authorization and jump right into implementing your ideas for new features. Rails Composer is often used at hackathons, where teams race to build interesting applications for a prize, to avoid the time sink of setting up a basic application.

A final word: Use Rails Composer judiciously. It's intended to be a tool for experienced developers who already know how to build starter applications from scratch using databases, front-end frameworks, authentication, or authorization, and all the bells and whistles offered in the Rails Composer options. Use it to pinpoint what you need to learn, or use it to turbocharge your learning process, but don't use it as a crutch to avoid learning the basics. To learn Rails, you must be able to build every starter application from scratch, without Rails

Composer.

To learn more about Rails Composer, see the Rails Composer home page and the README for the Rails Composer project on GitHub.

Chapter 26

Version Notes

If you are reading the online edition of the book, you have the most recent version of the book. If you've gotten your copy of the book elsewhere, you may have an older version that doesn't have the newest updates.

You'll find the version number and release date on the first page of this book (under the book title). Check the learn-rails GitHub repository to find out if you have the newest version of the book. The README page on the GitHub repo always shows the most recent version number for the book and the tutorial application.

If you have trouble building the application in this book, and suspect something may be out of date, you can check the Gemfile in the repo to see if we've changed gems or specified version numbers to avoid compatibility issues. You can also check the CHANGELOG, look at recent commits, and check the issues to see the current state of the application.

Here are the changes I've made.

Version 4.0.2

Version 4.0.2 was released November 25, 2016

Minor revisions for clarity.

Apple now calls the operating system macOS not Mac OS X.

Updated for Ruby 2.3.2.

Version 4.0.1

Version 4.0.1 was released November 4, 2016

Fixed broken links.

Removed references to Nitrous.io because Nitrous.io is out of business.

Version 4.0.0

Version 4.0.0 was released October 31, 2016

Updated for Rails 5.0 and Ruby 2.3.1.

Switch to using Bootstrap instead of Zurb Foundation front-end framework.

Extensive revisions.

Version 3.1.0

Version 3.1.0 was released March 1, 2016

Switch to using the SendGrid service to send email. Mandrill is no longer offering a free trial.

Version 3.0.2

Version 3.0.2 was released January 30, 2016

Minor change: 'email_provider_username' was 'mandrill_username' and email_provider_a
was 'mandrill_apikey'.

Version 3.0.1

Version 3.0.1 was released January 29, 2016

Specify version 5.5 of the foundation-rails gem. Foundation 6 is out but Zurb has not yet released documentation for migration from Foundation 5 to 6.

Version 3.0.0

Version 3.0.0 was released January 14, 2016

Extensive revision throughout the book, and the length of the book increased, so the book is now two books. Book One contains the introductory and self-help chapters and can be read without access to a computer. Book Two contains the step-by-step tutorial and requires use of a computer.

Switch to using the Mandrill service to send email. Previously used Gmail but Google has taken steps to make Gmail more secure and now it can be difficult to send email from a Rails application using Gmail.

Sending mail now requires the method `deliver_now` instead of `deliver`. The UserMailer class now inherits from `ApplicationMailer`.

Updated references to Rails from version 4.2.4 to 4.2.5.

Updated references to Ruby from version 2.2.3 to 2.3.0.

Version 2.2.2

Version 2.2.2 was released October 30, 2015

In the "Front-End Framework" chapter, updated filename to **`1st_load_framework.css.`** from **`framework_and_overrides.css.scss`** to reflect a change in the rails_layout gem.

Version 2.2.1

Version 2.2.1 was released September 19, 2015

Updated references to Ruby from version 2.2.0 to 2.2.3.

Updated references to Rails from 4.2.0 to Rails 4.2.4.

Updated Visitor model **`subscribe`** method for the new Gibbon 2.0 API.

Recommending Cloud9 instead of Nitrous.io because Nitrous.io is no longer free.

Version 2.2.0

Version 2.2.0 was released June 6, 2015

For Amazon customers, added an offer to access the online version or download a PDF at learn-rails.com.

Google now requires use of OAuth 2.0 for application access to Google Drive. The implementation is considerably more complex than the previous implementation using a Gmail address and password. I've dropped the "Spreadsheet Connection" chapter.

Minor clarification in the "Layout and Views" chapter.

Version 2.1.6

Version 2.1.6 was released March 17, 2015

Remove references to the Thin web server in the "Deploy" chapter.

Correct version number for **gem 'sass-rails'** in various Gemfile listings. Fixes issue 49 and an error "Sass::SyntaxError - Invalid CSS" when the Foundation front-end framework is used.

In the "Testing" chapter, the file *test/integration/home_page_test.rb* was missing **require 'test_helper'**.

Updated "Rails Composer" chapter to describe new options.

Minor improvements and corrections of typos.

Version 2.1.5

Version 2.1.5 was released March 4, 2015

Use the Ruby 1.9 hash syntax in the **validates_format_of :email** statement.

Minor improvements and corrections of typos.

Version 2.1.4

Version 2.1.4 was released January 3, 2015

Updated references to Ruby from version 2.1.5 to 2.2.0.

Specify the "v0" version of the google_drive gem in the "Spreadsheet Connection" chapter.

Version 2.1.3

Version 2.1.3 was released December 25, 2014

Updated references to Rails 4.1.8 to Rails 4.2.0.

Version 2.1.2

Version 2.1.2 was released December 4, 2014

Released for sale as a Kindle book on Amazon, with new cover art (same cat, though).

RailsApps Tutorials now named the Capstone Rails Tutorials.

Updated references to Ruby from version 2.1.3 to 2.1.5.

Updated references to Rails 4.1.6 to Rails 4.1.8 (minor releases with bug and security fixes).

Removed link to the (now defunct?) Lowdown web application in the "Plan Your Product" chapter.

Changes to the "Asynchronous Mailing" section of "Send Mail" chapter to describe Active Job in Rails 4.2.

Minor improvements to the "Dynamic Home Page," "Deploy," "Configure," "Troubleshoot," and "Create the Application" chapters.

Version 2.1.1

Version 2.1.1 was released October 22, 2014

Minor rewriting for clarity.

Updated "Precompile Assets" section of the "Deploy" chapter.

Mentioned explainshell.com in the "Get Started" chapter.

Mentioned Zeal as a Linux alternative to Dash.

Recommended book Practicing Rails by Justin Weiss.

Version 2.1.0

Version 2.1.0 was released October 12, 2014

Updated references to Ruby from version 2.1.1 to 2.1.3.

Updated references to Rails 4.1.1 to Rails 4.1.6 (minor releases with bug and security fixes).

Four new chapters:

- "Testing"
- "Rails Composer"
- "Crossing the Chasm"
- "Level Up"

Use ActiveModel instead of the activerecord-tableless gem.

In the "Configuration" chapter, add a note to use spaces (not tabs) in the **config/secrets.yml** file.

Updated "Gems" chapter to add a troubleshooting note to the "Install the Gems" section (about errors with the Nokogiri gem).

Added a section on "Multiple Terminal Windows" to the "Create the Application" chapter.

In the "Get Help When You Need It" chapter, updated the list of recommended newsletters, replaced rubypair.com with codermatch.me, and added a section on code review. Removed reference to defunct Rails Development Directory.

Version 2.0.2

Version 2.0.2 was released May 6, 2014

Updated references to Rails 4.1.0 to Rails 4.1.1 (a minor release with a security fix).

For Nitrous.io users, clarify that "http://localhost:3000/" means the Preview browser window.

Update "Gems" chapter, section "Where Do Gems Live?" to add more explanation.

Minor change to code in the "Mailing List" chapter, setting 'mailchimp_api_key' explicitly when instantiating Gibbon, for easier troubleshooting.

Version 2.0.1

Version 2.0.1 was released April 16, 2014

Minor updates for Rails 4.1.0. Mostly small changes to the "Configure" and "Front-End Framework" chapters.

Added an explanation that, in the **config/secrets.yml** file, `domain_name` doesn't have to be kept secret and set as a Unix environment variable.

Added a hint about passwords that use punctuation marks (plus a completely irrelevant note about profanitype).

Replaced `Rails.application.secrets.gmail_username` with `Rails.applicatio` Also replaced `gmail_password` with `email_provider_password`. Just trying to make things a little more generic in case Gmail is not used as a provider.

Added a section explaining the horrid details of the `config.assets.precompile` configuration setting in the **config/application.rb** file. Please convey my displeasure to those responsible for subjecting beginners to this travesty.

In the "Deploy" chapter, restored **RAILS_ENV=production rake assets:precompile** because Rails 4.1.0 no longer barfs on this.

Added resources to the "Get Help When You Need It" chapter.

Minor rewriting of the introduction.

Version 2.0.0

Version 2.0.0 was released April 8, 2014

Updated references to Ruby from version 2.1.0 to 2.1.1.

Updated the book to Rails 4.1. The application name is no longer used in the **config/routes.rb** file.

Rails 4.1 changes the **app/assets/stylesheets/application.css.scss** file. Updated the "Front-End Framework" chapter. Also expanded the explanation of the Foundation grid.

In Rails 4.1, configuration variables are set in the **config/secrets.yml** file. The Figaro gem is dropped, along with the **config/application.yml** file. Updated the "Configure" chapter and references to configuration variables throughout the book.

In the "Deploy" chapter, changed **RAILS_ENV=production rake assets:precompile** to **rake assets:precompile** to avoid the error "database configuration does not specify adapter."

Updated "The Parking Structure" chapter with comments about "Folders of Future Importance" that experienced developers often use: **test/**, **spec/**, **features/**, **policies/**, and **services/**. Updated the "Spreadsheet Connection" chapter to mention service-oriented architectures (SOA).

Extended the section on "Limitations of Metaphors" in the "Just Enough Ruby" chapter to include the example of gender when modeling a person.

Minor rewriting for clarity throughout.

Version 1.19

Version 1.19 was released February 1, 2014

Updated the book to use Foundation 5.0. Foundation 5.0.3 was released January 15, 2014 (earlier versions 5.0.1 and 5.0.2 were incompatible with Rails Turbolinks and the Rails asset pipeline). Changed the Gemfile to remove **gem 'compass-rails'** and replace **gem 'zurb-foundation'** with **gem 'foundation-rails'**. Updated a line in the "Front-End Framework" chapter for Foundation 5.0:

```
$ rails generate layout foundation5 --force
```

The files **navigation.html.erb** and **application.html.erb** are changed for Foundation 5.0. The Bootstrap front-end framework is now independent of Twitter, so I call it "Bootstrap" not "Twitter Bootstrap." Revised the chapter "Just Enough Ruby" to incorporate suggestions from technical editor Pat Shaughnessy. Revised the chapter "Request and Response" to incorporate suggestions from technical editor Kirsten Jones. Minor rewriting for clarity throughout.

Version 1.18

Version 1.18 was released January 10, 2014

Updated references to Ruby from version 2.0.0 to 2.1.0. Changed one line in the "Front-End Framework" chapter to accommodate a change in the rails_layout gem version 1.0.1. The command was:

```
$ rails generate layout foundation4 --force
```

Changed to:

```
$ rails generate layout:install foundation4 --force
```

Updated the "Configure" chapter to add ActionMailer configuration values to the file **config/environments/development.rb**.

Version 1.17

Version 1.17 was released December 21, 2013

Updated Rails version from 4.0.1 to 4.0.2 .

Changed Gemfile to remove **gem 'compass-rails', '> 2.0.alpha.0'** and replace it with **gem 'compass-rails', '> 1.1.2'**. The 2.0.alpha.0 version was yanked from the RubyGems server. The compass-rails gem is needed for Foundation 4.3. It will not be needed for Foundation 5.0.

Changed Gemfile to replace **gem 'zurb-foundation'** with **gem 'zurb-foundation' '> 4.3.2'**. Foundation 5.0 will require **gem 'foundation-rails'** but we can't use it until an incompatibility with Turbolinks is resolved. So we will stick with Foundation 4.3.2 for now.

Revised code in the "Analytics" chapter. Using **ready page:change** instead of **page:load** to accommodate Turbolinks. Updated the **segmentio.js** file to use a new tracking script from Segment.io. Updated instructions for setting up Google Analytics tracking on Segment.io. Added concluding paragraphs "Making Mr. Kadigan Happy" to the "Analytics" chapter.

Minor clarification in the "Front-End Framework" chapter to explain that the navigation bar won't show a dropdown menu until the next chapter, when we add navigation links.

Minor clarification in the "Spreadsheet Connection" chapter to explain that Google may block access if you attempt access from a new and different computer (including Nitrous.io).

Added cat names in the "Credits and Comments" chapter.

Revised "Getting Help" chapter and added "Version Notes" chapter.

Minor clarifications, plus fixes for various typos and insignificant errors.

Chapter 27

Credits and Comments

Was the book useful to you? Follow @rails_apps on Twitter and tweet some praise. I'd love to know you were helped out by the tutorial.

You can find me on Facebook or Google+. I'm happy to connect if you want to stay in touch.

If you'd like to recommend the book to others, the landing page for the book is here:

* http://learn-rails.com/learn-ruby-on-rails.html

I'd love it if you mention the book online, whether it is a blog post, Twitter, Facebook, or online forums. Recommending the book with a link makes it easier for people to find the book.

Credits

The book was created with the encouragement, financial support, and editorial assistance of hundreds of people in the Rails community.

Daniel Kehoe wrote the book and implemented the application.

Kickstarter

Thank you to contributors to two Kickstarter campaigns, for the first edition of the book and for the revision for Rails 5.0.

- Learn Ruby on Rails

- Learn Ruby on Rails 5

Financial Backers for the First Edition

The following individuals provided financial contributions of over $50 to help launch the book. Please join me in thanking them for their encouragement and support.

Al Zimmerman, Alan W. Smith, Alberto A. Colón Viera, Andrew Terry, Avi Flombaum, Brian Hays, Charles Treece, Dave Doolin, Denzil Villarico, Derek Rockwell, Eito Katagiri, Evan Sparkman, Frank Castle, Fred Dixon, Fred Schoeneman, Gant Laborde, Gardner Monks, Gerard de Brieder, GoodWorksOnEarth.org, Hanspeter Leupin, Harald Lazardzig, Harsh Patel, James Bond, Jared Koumentis, Jason Landry, Jeff Whitmire, Jesse House, Joe Wilmoth Jr., John Shannon, Joost Baaij, Juan Cristobal Pazos, Kathleen Sidenblad, Laird Hayward, Logan Hasson, Ludovic Kuty, Mark Gilbert, Matt Esterly, Mike Gilbert, Niko Roberts, Norman Cohen, Paul Philippov, Robert Nadar, Rogier Hof, Ross Kinney, Ruben Calzadilla, Stephane Moreau, Susan Wilson, Sven Fuchs, Thomas Nitsche, Tom Michel, Youn Shin Kang, Yuen Lock

Technical Editors

Rails and Ruby experts are very busy. I am very grateful for the assistance I received from my colleagues for the technical review of individual chapters.

- Kirsten Jones, reviewed the chapter "Request and Response"

- Pat Shaughnessy, author of Ruby Under a Microscope, reviewed the chapter "Just Enough Ruby"

- Noel Rappin, author of Rails Test Prescriptions, reviewed chapters 1-7, and the "Testing" chapter

- Aaron Sumner, author of Everyday Rails Testing with RSpec, reviewed the "Testing" chapter

- Ken Collins reviewed the "Testing" chapter

Buy their books. I recommend them.

Editors and Proofreaders

Dozens of volunteers offered corrections and made suggestions, from fixing typos to advice about organizing the chapters.

Alberto Dubois Ribó, Alex Finnarn, Alex Zielonko, Alexandru Muntean, Alexey Dotokin, Alexey Ershov, André Arko, Andreas Basurto, Ben Swee, Brandon Schabel, Cam Skene, Daniella Zimmermann, Dapo Babatunde, Dave Levine, Dave Mox, David Kim, Duany Dreyton Bezerra Sousa, Erik Trautman, Erin Nedza, Flavio Bordoni, Fritz Rodriguez Jr, Hendri Firmana, Ishan Shah, James Hamilton, Jasna Vukovic, Jeremy Schneider, Joanne Daudier, Joel Dezenzio, Jonah Ruiz, Jonathan Lai, Jonathan Miller, Jordan Stone, Joreal Whitfield, Josh Morrow, Joyce Hsu, Julia Mokus, Julie Hamwood, Jutta Frieden, Laura Pierson Wadden, Marc Ignacio, Mark D. Blackwell, Mark Everhart, Michael Wong, Miguel Herrera, Mike Janicki, Miran Omanovic, Neha Jain, Norman Cohen, Oana Sipos, Peter Rangelov, Richard Afolabi, Robin Paul, Roderick Silva, Sakib Ash, Sebastian Lobato Genco, Silvia Obajdin, Stas Sucov, Stefan Streichsbier, Sven Fuchs, Tam Eastley, Tim Goshinski, Timothy Jones, Tom Connolly, Tom Michel, Tomas Olivares, Verena Brodbeck, Will Schive, William Yorgan, Zachary Davy

Photos

Images provided by the lorempixel.com service are used under the Creative Commons license. Visit the Flickr accounts of the photographers to learn more about their work:

- photo of a white cat by Tomi Tapio

- photo of a cat by Steve Garner

- photo of a cat by Ian Barbour

The photo of a fluffy white cat by Tomi Tapio is used in the application.

Comments

I regularly update the book. Your comments and suggestions for improvements are welcome.

Feel free to email me directly at daniel@danielkehoe.com.

Are you stuck with code that won't work? Stack Overflow provides a question-and-answer forum for readers of this book. Use the tag "learn-ruby-on-rails" or "railsapps" when you post your question.

Found a bug in the tutorial application? Please create an issue on GitHub.