learn-rails.com

Learn Ruby on Rails

A tutorial by Daniel Kehoe · 1.b15 (prerelease) · 3 November 2013

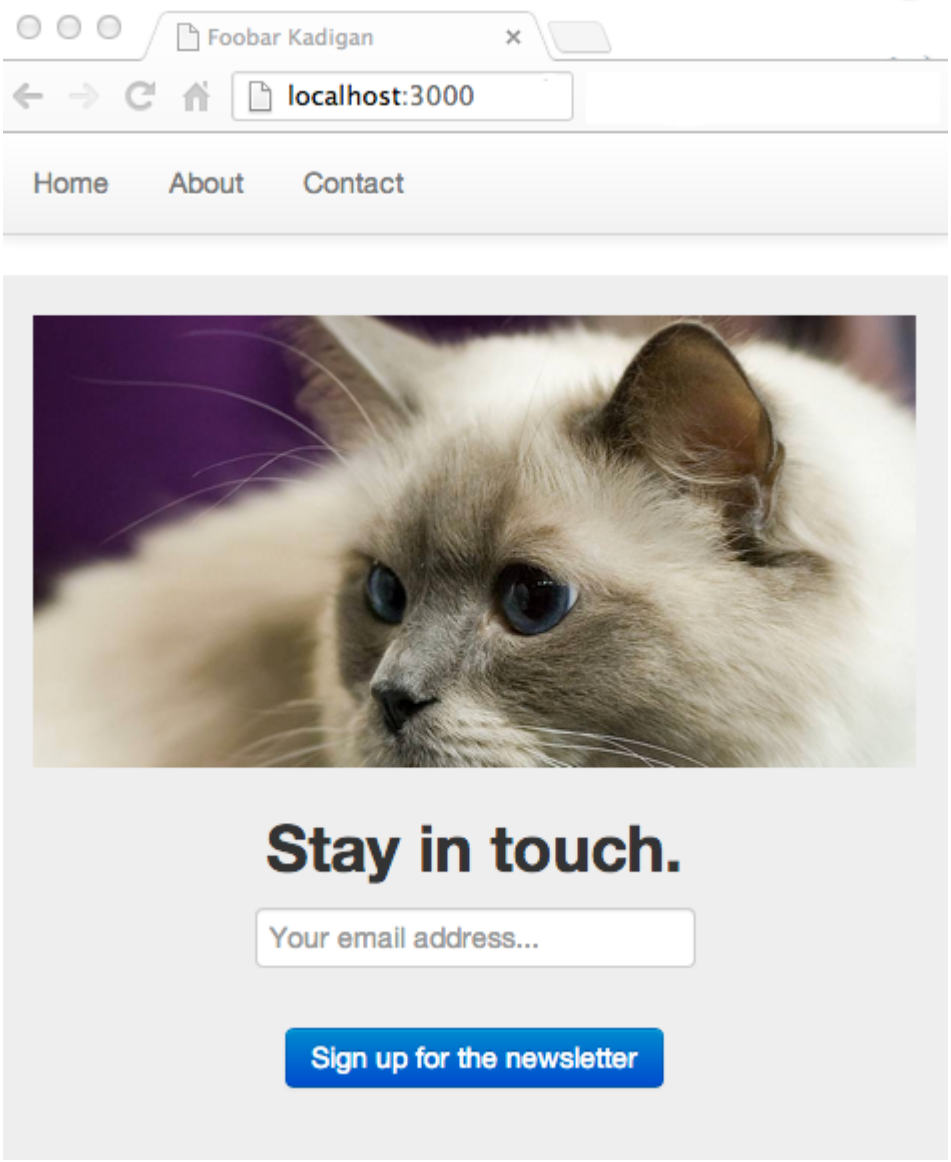
Contents

Chapter 1

Introduction

Welcome. This tutorial is a first step on your path to learn Ruby on Rails.

You'll learn key concepts so you'll have a solid foundation for continued study. You'll build a working web application so you'll gain hands-on experience. Along the way, you'll practice the techniques used by professional Rails developers and you'll understand why Rails has become a popular choice for web development.



Is It for You?

You don't need to be a programmer to succeed with this tutorial. You'll get comfortable with the Ruby programming language and the Unix command line interface as you build a Rails application.

This tutorial is ideal if you are:

- a student
- a startup founder
- making a career change

Does this sound like you? Readers who work in social media or graphic design say this tutorial is a good way to get introduced to programming. Others who previously built simple websites using HTML, or used applications such as WordPress, found they could easily progress to building websites with Rails. Programmers with experience in languages such as PHP or Java found this tutorial to be a good way to get started with the Rails framework.

On the other hand, if you've never encountered HTML, it is best to start elsewhere with an "Introduction to Web Design" course or online tutorial.

Warnings

This is two books in one. At the core is a hands-on tutorial that will lead you through the code needed to build a real-world web application. I thoroughly explain the code you'll need to write a Rails application. Before you start coding, I explain the culture and practices of the Rails community. If you're in a hurry to start coding, jump right now to Chapter 7, "Get Started." But I urge you to read the preliminary chapters first. They'll give you the context you need to become a skilled Rails developer. Many readers have told me the concepts at the beginning of the book provide a grounding they haven't found in any other tutorial.

The tutorial is designed to unfold in steps, one section leading to another. You can use the book as a reference, skipping around without reading from beginning to end, but you'll actually waste time as you try to pick up the pieces you missed.

The chapters are densely packed with links to 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.

There is rich satisfaction in building something and making it run. But programming can be frustrating and Rails isn't easy for beginners. Before you get to the reward, you'll encounter setbacks. If at times you're ready to quit, jump to the chapter titled "Rails Challenges"at the

end of the book. It describes many of the problems learners encounter. I've written it to address your concerns when learning Rails becomes difficult and frustrating.

It's best to work through the book from start to end, allowing enough time to read the introductory chapters and then building the application. That means you should allow time to read the book before you start a new job or join a developer bootcamp. Really!

What To Expect

When you've completed this tutorial, you will be ready for more advanced self-study, including other tutorials from the RailsApps project, textbook introductions to Rails, or workshops and developer bootcamps that provide intensive training in Ruby on Rails. Other curriculums often skip the basics. With this tutorial you'll have a solid grounding in key concepts; you won't feel overwhelmed or frustrated as you continue your studies.

This tutorial is good preparation for:

- advanced tutorials from the RailsApps Project
- textbooks such as Michael Hartl's Ruby on Rails Tutorial
- introductory workshops from RailsBridge or Rails Girls
- intensive training with immersive code camps

We are blessed with many textbooks, workshops, and classroom programs that teach Ruby on Rails. I believe this book is unique in covering the basics while introducing the tools and techniques of professional Rails development.

The RailsApps Project

This book is the foundation for a series of tutorials that accompany example applications from the RailsApps project.

Tutorials from @rails_apps take you on a guided path starting with absolute basics (this tutorial). You'll progress to intermediate-level tutorials and soon be using the RailsApps in-depth guides for professional Rails developers.

It is important to feel satisfaction and accomplishment as you learn. That's why each tutorial introduces Rails in stages. With each tutorial you will build a real-world Rails application. The finished product confirms your accomplishment; you'll feel genuine satisfaction as you deploy your Rails application. Hands-on learning with real Rails applications is the key to absorbing and retaining knowledge.

The applications you'll build in the tutorials are not classroom exercises. The primary purpose of the RailsApps project is to provide starter applications for Rails developers.

You'll build real applications that you can customize and adapt for your startup, at your job, or for clients.

Hundreds of developers use the RailsApps example applications, report problems as they arise, and propose solutions. Rails changes frequently; each application is known to work and serves as your personal "reference implementation" so you can stay up to date. Maintenance and development of the RailsApps applications is supported by subscriptions to the RailsApps tutorials.

The Application

We'll build a basic web application that can be used by a typical small business. The website will include a home page, "about" page, contact form, and option to sign up for a mailing list. You'll also learn how to collect data from a form and save it to a spreadsheet on Google Drive.

You'll find the complete learn-rails application on GitHub. It is a working application that is maintained by a team of experienced developers so you can always check the "reference implementation" if you have problems.

A Note to Reviewers and Teachers

This book approaches the subject differently than most introductions to Rails. It introduces concepts of product planning, project management, and website analytics to place development within a larger context of product development and marketing. In introducing Rails, rather than show the student how to use scaffolding, it introduces the model-view-controller design pattern by creating the components manually. The tutorial recommends test-driven development, but doesn't show it, simply because I've found TDD can't be adequately covered in a basic introduction. 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 developer bootcamp 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 required reading, contact me at daniel@danielkehoe.com to arrange access to the book for your students. The book is available at no charge to students enrolled in free workshops or classes, thanks to generous gifts from prominent members of the Rails community.

Chapter 2

Concepts

This chapter provides the background, or "big picture," you will need to understand Rails.

This chapter is excerpted from an in-depth article What is Ruby on Rails? For a deeper understanding of Rails, including background on the guiding principles of Rails, and reasons for its popularity, read the article for a complete introduction.

Here are the key concepts you'll need to know before you try to use Rails.

How the Web Works

We start with absolute basics, as promised.

When you "visit a website on the Internet" you use a *web browser* such as Safari, Chrome, Firefox, or Internet Explorer.

Web browsers are *applications* (software programs) that work by reading *files*.

Compare a *word processing program* with a *web browser*. Both word processing programs and web browsers read files. Microsoft Word reads files that are stored on your computer to display documents. A web browser retrieves files from remote computers called *servers* to display web pages. Knowing that everything comes from files will help you build a web application.

A web browser uses four kinds of files to display web pages:

- HTML – *structure* (layout) and *content* (text)
- CSS – *stylesheets* to set visual appearance
- JavaScript – *programming* to alter the page
- Images

At a minimum, a web page requires an HTML file. If a web browser receives just an HTML file, it will display text, with default styles applied by the browser.

If the page is always the same, every time it is displayed by the web browser, we say it is *static*. Webmasters don't need software such as Rails to deliver static documents; they just create files for delivery by an ordinary *web server* program.

Static websites are ideal for particle physics papers (which was the original use of the World Wide Web). But most sites on the web, especially those that allow a user to sign in, post comments, or order products and services, generate web pages *dynamically*.

Dynamic websites often combine web pages with information from a database. A database stores information such as a user's name, comments, Facebook likes, advertisements, or any other repetitive, structured data. A database *query* can provide a selection of data that customizes a webpage for a particular user or changes the web page so it varies with each visit.

Dynamic websites use a programming language such as Ruby to assemble HTML, CSS, and JavaScript files on the fly from component files or a database. A software program written in Ruby and organized using the Rails *development framework* is a Rails *web application*. A web server program that runs Rails applications to generate dynamic web pages is an *application server* (but usually we just call it a web server).

Software such as Rails can access a database, combining the results of a database query with static content to be delivered to a web browser as HTML, CSS, and JavaScript files. Keep in mind that the web browser only receives ordinary HTML, CSS, and JavaScript files; the files themselves are assembled dynamically by the Rails application running on the server.

Even if you are not going to use a database, there are other good reasons to generate a website using a programming language. For example, if you are creating several web pages, it often makes sense to assemble an HTML file from smaller components. For example, you might make a small file that will be included on every page to make a footer (Rails calls these "partials"). Just as importantly, if you are using Rails, you can add features to your website with code that has been developed and tested by other people so you don't have to build everything yourself.

The widespread practice of sharing code with other developers for free, and collaborating with strangers to build applications or tools, is known as *open source* software development. Rails is at the heart of a vibrant open source development community, which means you leverage the work of tens of thousands of skilled developers when you build a Rails application. When Ruby code is packaged up for others to share, the package is called a *gem*. The name is apt because shared code is valuable.

Ruby is a programming language; Rails is a development framework. That means Rails is a set of *structures and conventions* for building a web application using the Ruby language. Rails is also a *library* or collection of *gems* that developers use as the core of any Rails web application. By using Rails, you get well-tested code that implements many of the most-needed features of a dynamic website.

With Rails, you will be using shared standard practices that make it easier to collaborate with others and maintain your application. As an example, consider the code that is used to access a database. Using Ruby without the Rails framework, or using another language such as PHP, you could mix the complex programming code that accesses the database with the code that generates HTML. With the insight of years of developers' collective experience in maintaining and debugging such code, Rails provides a library of code that segregates

database access from the code that displays pages, enforcing *separation of concerns*, and making more modular, maintainable programs.

In a nutshell, that's how the web works, and why Rails is useful.

For a history of Rails, and an explanation of why it is popular, see the article What is Ruby on Rails?

What is Rails?

So far, I've defined Rails in two ways: as *structures and conventions* for building a web application, and as a *library* or collection of code.

To really understand Rails, and succeed in building Rails applications, we need to consider Rails from six other perspectives. Like six blind men encountering an elephant, it can be difficult to understand Rails unless you look at it from multiple points of view.

Here are six different ways of looking at Rails, summarized from the article What is Ruby on Rails?

From the **perspective of the web browser**, Rails is simply a program that generates HTML, CSS, and JavaScript files. These files are generated dynamically. You can't see the files on the server side but you can view these files by using the web developer tools that are built in to every browser. Later you'll examine these files when you learn to troubleshoot a Rails application.

From the **perspective of a programmer**, Rails is a set of files organized with a specific structure. The structure is the same for every Rails application; this commonality is what makes it easy to collaborate with other Rails developers. We use text editors to edit these files to make a web application.

From the **perspective of a software architect**, Rails is a structure of *abstractions* that enable programmers to collaborate and organize their code. Thinking in abstractions means we group things in categories and analyze relationships. Conceptual categories and relationships can be made "real" in code. Software programs are built of "concepts made real" that are the moving parts of a software machine. To a software architect, *classes* are the basic parts of a software machine. A class can represent something in the physical world as a collection of various attributes or properties (for example, a User with a name, password, and email address). Or a class can describe another abstraction, such as a Number, with attributes such as quantity, and behavior, such as "can be added and subtracted." You'll get a better grasp of classes in the chapter, "Just Enough Ruby."

To a software architect, Rails is a pre-defined set of classes that are organized into a higher level of abstraction known as an API, or *application programming interface*. The Rails API is organized to conform to certain widely known *software design patterns*. You'll become familiar with these abstractions as you build a Rails application. Later in the tutorial, we'll

learn about the *model–view–controller* design pattern. As a beginner, you will see the MVC design pattern reflected in the file structure of a Rails application.

We can look at Rails from the **perspective of a gem hunter**. Rails is popular because developers have written and shared many software libraries (RubyGems, or "gems") that provide useful features for building websites. We can think of a Rails application as a collection of gems that provide basic functionality, plus custom code that adds unique features for a particular website. Some gems are required by every Rails application. For example, database adaptors enable Rails to connect to databases. Other gems are used to make development easier, for example, gems for testing that help programmers find bugs. Still other gems add functionality to the website, such as gems for logging in users or processing credit cards. Knowing what gems to use, and why, is an important aspect of learning Rails. This tutorial will show you how to build a web application using some of the most commonly used gems.

We can also look at Rails from the **perspective of a time traveler** in order to understand the importance of *software version control*. Specifically, we use the Git revision control system to record a series of snapshots of your project's filesystem. Git makes it easy to back up and recover files; more importantly, Git lets you make exploratory changes, trying out code you may decide to discard, without disturbing work you've done earlier. You can use Git with GitHub, a popular "social coding" website, for remote backup of your projects and community collaboration. Git can keep multiple versions ("branches") of your local code in sync with a remote GitHub repository, making it possible to collaborate with others on open source or proprietary projects. Strictly speaking, Git and GitHub are not part of Rails (they are tools that can be used on any development project). And there are several other version control systems that are used in open source development. But a professional Rails developer uses Git and GitHub constantly on any real-world Rails project. Rails and the gems that go into a complex web application would not exist without Git and GitHub.

Finally, we can consider a Rails application from the **perspective of a tester**. Software testing is part of Rails culture; Rails is the first web development platform to make testing an integrated part of development. Before Rails, automated testing was rarely part of web development. A web application would be tested by users and (maybe) a QA team. If automated tests were used, the tests were often written after the web application was largely complete. Rails introduced the discipline of Test-Driven Development (TDD) to the wider web development community. With TDD, tests are written *before* any implementation coding. It may seem odd to write tests first, but for a skilled TDD practitioner, it brings coherence to the programming process. First, the developer will give thought to what needs to be accomplished and think through alternatives and edge cases. Second, the developer will have complete test coverage for the project. With good test coverage, it is easier to *refactor*, rearranging code to be more elegant or efficient. Running a test suite after refactoring provides assurance that nothing inadvertently broke after the changes.

TDD is seen as a necessary skill of an experienced Rails developer. Because this is a tutorial for beginners, it *will not* introduce you to techniques of Test-Driven Development. As you work through more advanced tutorials, you'll be introduced to Test-Driven Development.

Stacks

To understand Rails from the perspective of a professional Rails developer, you'll need to grasp the idea of a *technology stack* and recognize that Rails can have more than one stack.

A technology stack is a set of technologies or software libraries that are used to develop an application or deliver web pages. "Stack" is a term that is used loosely and descriptively. There is no organization that sets the rules about what goes into a stack. As a technologist, your choice of stack reflects your experience, values, and personal preference, just like religion or favorite beverage.

For example, Mark Zuckerberg developed Facebook in 2004 using the LAMP application stack:

- Linux (operating system)
- Apache (web server)
- MySQL (database)
- PHP (programming language)

For this tutorial, your application stack will be:

- Mac OS X, Linux, or Windows
- WEBrick (web server)
- SQLite (database)
- Ruby on Rails (language and framework)

Sometimes when we talk about a stack, we only care about part of a larger stack. For example, a Rails stack includes the gems we choose to add features to a website or make development easier. When we select the gems we'll use for a Rails application, we're choosing a stack.

Sometimes the choice of components is driven by the requirements of an application. At other times, the stack is a matter of personal preference. Just as craftsmen and aficionados debate the merits of favorite tools and techniques in any profession, Rails developers avidly dispute what's the best Rails stack for development.

The company 37signals, where the creator of Rails works, uses this Rails stack:

- ERB for view templates
- MySQL for databases
- MiniTest for testing

It is not important (at this point) to know what the acronyms mean (we'll learn later).

Another stack is more popular among Rails developers:

- Haml for view templates
- PostgreSQL for databases
- Rspec for testing

We'll learn later what the terms mean. For now, just recognize that parts of the Rails framework can be swapped out, just like making substitutions when you order from a menu at a restaurant.

You can learn much about Rails by following the experts' debates about the merits of a favorite stack. The debates are a source of much innovation and improvement for the Rails framework. In the end, the power of the crowd prevails; usually the best components in the Rails stack are the most popular.

The proliferation of choices for the Rails stack can make learning difficult, particularly because the components used by many leading Rails developers are not the components used in many beginner tutorials. In this tutorial, we stick to solid ground where there is no debate. In more advanced tutorials, we'll explore stack choices and choose components that are most often used by professional developers.

Chapter 3

Get Help When You Need It

I'm often asked, "Where's the Rails manual?" There isn't one. No single document tells you how to use Rails. Instead, there's a wealth of documentation that describes various aspects of Rails. You won't need any other documentation to complete this tutorial but I'd like to suggest some resources that will be helpful as you go deeper in your study of Rails.

Getting Help

First of all, what to 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.

Rails Hotline is a free telephone hotline for Rails questions staffed by volunteers. You'll need to carefully think about and describe your problem but sometimes there's no better help than a live expert.

References

In addition to the resources listed here, the RailsApps project offers a list of top resources for Ruby and Rails, including books and blogs.

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.

Here are suggestions for the most important additional references.

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.

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
- Rails Searchable API Doc – alternative interface for the API docs
- apidock.com/rails – Rails API docs with usage notes
- apidock.com/ruby – Ruby API docs with usage notes

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.

The best source of news is Peter Cooper's Ruby Weekly email newsletter. It arrives each Thursday and it is free. For more frequent news, check Peter Cooper's RubyFlow site which lists new blog posts from Rails developers each day.

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

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 rubypair.com or letspair.net.

Let's look at some social aspects of product development before we get into building an application.

Chapter 4

Plan Your Product

Tutorials from other authors focus only on coding. But Rails developers do more than code. Software development is a process that begins with planning and ends with analysis and review. Coding, testing, and deployment is at the core but you'll need to learn about the entire process to succeed professionally. That's why we look at product planning and project management.

For this beginning tutorial, we'll introduce concepts about product planning and project management that you will encounter as a Rails developer. If you are interested in diving deeper, see the article Rails and Product Planning.

Product Owner

On your project, who is the *product owner*?

The product owner is the advocate for the customer, making sure that the team creates value for the users.

If you are a solo operator, you are the one who will decide what features and functionality will be included in your application. But if you're part of a team, either in a startup, as a consultant, or in a corporate setting, it may not be clear who has responsibility for looking at the application from the point of view of the application user. Someone must decide which features and functionality are essential and which must be left out. We call this *managing scope* and combating *feature creep*.

It's important to assign a product owner. Without a product owner in charge, tasks remain vague and developers have difficulty making progress.

In large organizations, a product owner may be a product manager or a project manager. A product owner usually is not a management executive (though there will likely be an executive sponsor). Everyone on the team — including management, developers, and stakeholders — should agree to designate a product owner and give that person authority to define features and requirements.

User Stories

A product owner's principal tool for product planning is the *user story*.

In the past, when software engineering primarily served government or large corporations, product planning started with *requirements gathering* defined as *use cases*, and culminated in a *requirements specification*. User stories are a faster, more flexible approach to product planning.

User stories are a way to discuss and describe the requirements for a software application. The process of writing user stories helps a product owner identify all the features that are needed for an application. Breaking down the application's functionality into discrete user stories helps organize the work and track progress toward completion.

User stories are often expressed in the following format:

```
As a <role>
I want <goal>
In order to <benefit>
```

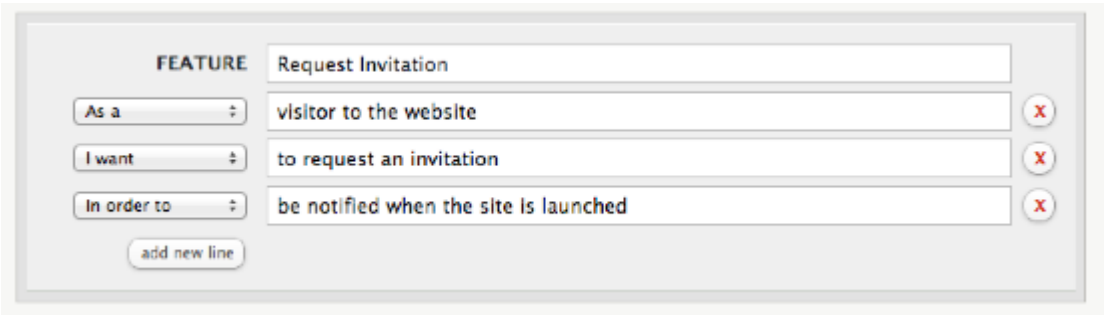
Here is an example:

```
*Join Mailing List*
As a visitor to the website
I want to join a mailing list
In order to receive news and announcements
```

A typical application has dozens of user stories, from basic sign-in requirements to the particular functionality that makes the application unique.

You don't need special software to write user stories. Just use index cards or a Word document. In the next chapter, we'll see how you can enter user stories as tasks in a to-do list.

Here's a screenshot from Lowdown, a web application that developers use for organizing user stories.



Just like Rails provides a structure for building a web application, user stories provide a structure for organizing your product plan.

Wireframes and Mockups

Often, before writing user stories, a product owner will make rough sketches of various web pages. Sketching is a phase where you try out ideas to clarify your vision for the application. Sketching can lead to a wireframe or a mockup. These terms are often used interchangeably but there are differences in meaning.

A *wireframe* is a drawing showing all functional elements of a web page. It should not depict a proposed graphic design for a website, rather it should be a diagram of a web page, without color or graphics.

A *mockup* adds graphic design to a wireframe; including branding devices, color, and placeholder content. A mockup gives an impression of the website's "personality" as well as proposed functionality.

One of the most popular tools for creating wireframes is Balsamiq Mockups (despite the name, it produces wireframes, not mockups). There are dozens of others listed in the article Rails and Product Planning.

As a product owner, writing user stories or sketching wireframes will help you refine product requirements. Some people like a visual approach with wireframes; others prefer words and narrative. Either approach will work; both are good.

Graphic Design

Very few people have skills as both a visual designer and a programmer. The tools are different; graphic designers typically use Adobe Photoshop, though web-savvy designers often create designs directly in HTML and CSS, while developers write code.

If you're lucky, you will work with a skilled graphic designer as you build your web application. If you are very lucky, you may work with someone who is a *user experience* (UX) designer or *interaction designer* (IxD). Interaction design is a demanding, sophisticated discipline that requires the mindset of an anthropologist and the eye of a visual artist to find not just the most pleasing, but the most effective visual design for an application user interface. You can find interaction designers discussing their concerns on the IxDA website, including the differences between interaction design and UX design.

If you're working with a graphic designer you might collaborate on a *moodboard* or a *design brief* to define the look and feel of your application. If the designer works in Photoshop, you'll face the challenge of converting design layouts from Photoshop to HTML and CSS. There are service firms that do this for a fee but obviously it's easier to work with a designer who can implement a layout directly in HTML and CSS.

Rails can be particularly challenging when it comes to integrating graphic design with code. Rails uses a hybrid of HTML markup mixed with Ruby programming code in its *view* files

(depending on the stack you've selected, the view files can use ERB, Haml, or other syntaxes for mixing HTML and Ruby). Few designers are comfortable with Ruby code mixed with HTML so you may end up doing integration yourself.

If you don't have a skilled graphic designer available to help, you can use Twitter Bootstrap or other front-end frameworks such as Zurb Foundation to quickly add an attractive design to your application.

You can use DivShot, a drag-and-drop interface builder that uses Twitter Bootstrap for layout and exports HTML and CSS code ready to integrate with your Rails application. DivShot was built by an experienced Rails developer; Bootstrap Designer, Bootply, and Jetstrap are similar tools.

Software Development Process

Product planning is the initial phase of a larger software development process. You can approach this casually, and start coding with curiosity and ambition, finding your own best way to the end product, by trial and error. Most hobbyist and student developers need no other approach.

When money or reputation is at stake, casual approaches to software development are risky. Compared to other forms of engineering, software development is peculiarly prone to failure. As recently as 2003, IBM stated, "Most software projects fail. In fact, the Standish group reports that over 80% of projects are unsuccessful either because they are over budget, late, missing function, or a combination. Moreover, 30% of software projects are so poorly executed that they are canceled before completion."

Professional software developers, being intelligent and reflexive, and driven by a desire to become more efficient, or wanting to avoid the wrath of bosses and clients, frequently look for ways to reduce risk and improve the software development process. In recent years they've succeeded in improving the success rate of software engineering, largely due to the adoption of *software development methodologies* that improve the *business process* of producing software.

If you're a hobbyist or casual programmer, you don't need to learn about software development methodologies.

If you are going to be held accountable for the success or failure of a project, you should learn more about software development methodologies.

If you're going to be interviewing for a job as a programmer, it pays to recognize some of the names of software development methodologies and ask whether your employer has adopted a particular approach, especially if you'd like to work for a company that prides itself on being well-organized and supportive of staff development. Hiring managers may say, "we've synthesized several methodologies," which may mean they are ignorant or don't give a damn, or it may mean they are prepared to thoughtfully discuss the merits of various

approaches to software development. Managers who can discuss software development methodologies are more likely to be concerned about the welfare of their team.

Here are some software development methodologies you may hear about, with some notable characteristics:

- waterfall process – an old and disparaged approach
- Agile software development – an iterative and incremental approach
- Scrum – known for "sprints" and daily standup meetings
- Extreme Programming – pair programming and test-driven development

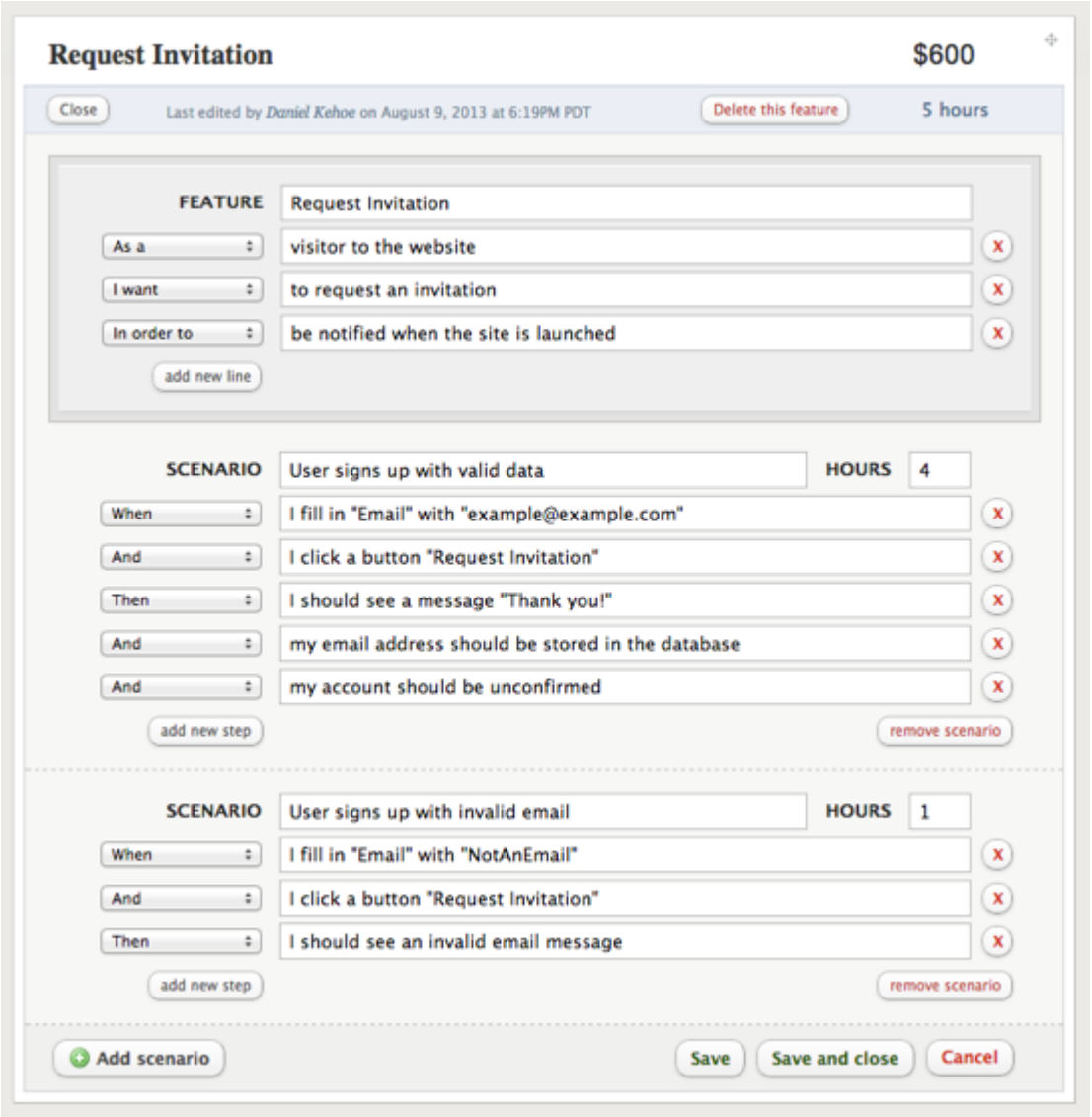
As you mature as a software developer, take time to think about the process of building software and learn more about software development methodologies.

Behavior-Driven Development

There is one prominent software development methodology that is important for product planning. It is called Behavior-Driven Development (BDD), or sometimes, Behavior-Driven Design.

BDD takes user stories and turns them into detailed scenarios that are accompanied by tests.

Here's a screenshot from the Lowdown web application that shows how a user story can be extended from a "feature" to include detailed "scenarios."



Rails developers turn these scenarios into tests and use a software tool named Cucumber to run automated test suites.

With automated tests, a product owner can determine if developers have succeeded in implementing the required features. This process is called *acceptance testing*. Automated tests also make it easy for developers to determine if the application still works as they add features, fix bugs, or reorganize code. This process is called *regression testing*.

On a small project like our tutorial application, you won't use BDD or Cucumber. It's easy enough to manually test the features before you deploy it.

For an introductory book, BDD is an advanced topic. But on a project where money and reputation is at stake, BDD can be very important. Every time an application is deployed, there's a chance that something could be broken. Software development is plagued with "fix one thing, accidentally break another" as code is refactored or improved. Manual testing can't be expected to reveal every bug. That's why automated testing, providing coverage of

every significant user-facing feature, is the only way to know if you've deployed without known bugs.

Chapter 5

Manage Your Project

How do you know you're making progress? Are you taking care of everything that needs to be done? These questions are at the center of project management. Whether you are working alone or as part of a team, you need to define your tasks and track progress toward your goal.

The previous chapter on product planning showed how *user stories* can be used to break down an application into discrete features. User stories can be the basis for a list of tasks.

To-Do List

You can track your tasks with a simple to-do list. Some entrepreneurs like the discipline of the GTD system (Getting Things Done) for personal productivity and time management. Our article on Rails and Project Management offers a list of popular to-do list applications, either for personal task management or team-oriented task management.

Kanban

Kanban is a method of managing projects that has been adapted from lean manufacturing for use in software development. In Japanese, "Kan" means visual, and "ban" means card or board.

Imagine putting a big whiteboard on your wall and creating columns for a series of to-do lists. The columns, called *swimlanes*, are labelled: Backlog, Ready, Coding, Testing, Done. Each swimlane contains index cards that describe a user story or other task. To plan your work and track progress, you'll move the index cards across the board from column to column. To stay focused and avoid becoming overwhelmed, you'll only pick the most important user stories or tasks from the backlog column and you'll limit the number of items in each column to what can be realistically accomplished in the time available. That's the essence of kanban as it is used for software development.

See the article on Rails and Project Management for a list of kanban web applications. Trello is particularly popular for task management.

Agile Methodologies

For a solo project or a small team, you'll do fine with a simple to-do list or (even better) a kanban web application for managing your software development process.

If you've got enough people to need to hire a project manager, you should look at project management software that supports teams using Agile software development methodologies. Pivotal Tracker is the best known tool but there are many other agile tools.

Learn more about Agile if you're going hire developers for a startup or if you are going to work for an established company. In most successful companies, Agile processes have replaced the much-maligned waterfall process that was once the norm for software development.

Our article on Rails and Project Management goes into more detail.

Chapter 6

Accounts You May Need

This tutorial will show you how to save your work using GitHub. You can sign up for a GitHub account for free.

We'll also send email from the application, and save data to Google Drive, which will require a Gmail account. A Gmail account is free.

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.

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.

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.

Gmail

The tutorial shows how the application can connect to a Gmail account to send email. We use Gmail as our example because many people already have a Gmail account. We will use your Gmail username and password to save data to Google Drive. You can get a free Gmail account if you don't already have one.

Some Google accounts require 2-step verification, which sends a unique code to your mobile phone each time you log in from an unfamiliar device. If your Google account requires two-factor authentication, you have two choices:

- turn off 2-step verification
- create a new Gmail account for use with this tutorial

Other services, such as Mandrill, can be used to send email from the application. Or you can connect directly to an SMTP mail server to send email. The tutorial won't show the details but I'll provide links for more information if you don't want to use Gmail.

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.

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.

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://id.heroku.com/signup/devcenter 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 7

Get Started

Before You Start

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.

If you find problems or wish to suggest improvements, it's best to create a GitHub issue. Feel free to email me directly at daniel@danielkehoe.com, but opening a GitHub issue will get you help from the larger community.

Your Computer

Mac OS X, Linux, or Windows

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

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 Nitrous.io, a browser-based development environment, on your Windows laptop.

Hosted Computing

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

The Nitrous.io 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 Nitrous.io, including a tablet or smartphone, though you need a broadband connection, a sizeable screen, and a keyboard to be productive. If you are using Windows, or have difficulty installing Ruby on your computer, try using Nitrous.io.

Text Editor

You'll need a text editor for writing code and editing files. I recommend Sublime Text 2 for Mac OS X, Windows, or Linux.

Word processing programs, such as Microsoft Word, will not work because they introduce hidden formatting codes into text files.

Programmers' text editors, such as Sublime Text, provide *syntax highlighting*, making software code more readable and programmers more productive. Simple text editors such as TextEdit for Mac OS X, or WordPad for Microsoft Windows, provide no syntax highlighting and should be avoided.

If you don't have a text editor, install Sublime Text now. You can find tutorials for Sublime Text on YouTube, including a popular Sublime Text 2 video from NetTutsPlus. It is not practical to explain how to set up and use a text editor in this short book, so use the instructions you'll find elsewhere.

You Don't Need an IDE

Programmers who come to Rails from other platforms, such as Java or C++, often ask for recommendations for an IDE, or an integrated development environment. These are software applications that combine a text editor with built-in tools such as a debugger. Some Rails developers use JetBrains RubyMine, Aptana Studio, or Komodo but most Rails developers use only a text editor and terminal application. You don't need an IDE unless you're in the habit of using one. For a beginner, they are cumbersome and add little additional value.

Terminal

You'll need an application called a console or terminal emulator to run programs from your computer's command line. We call the command line the *shell* because it is the outer layer of the operating system's internal mechanisms (which we call the *kernel*).

On Mac OS X, you can use the Terminal application. Experienced developers often upgrade to the more powerful iTerm2 application.

The Command Line Crash Course explains how to launch a terminal application.

If you haven't used the computer's command line interface (CLI) before, spend some time with The Command Line Crash Course to become comfortable with Unix shell commands.

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.

Getting Fancy

If you watch experienced developers at work, you may see their consoles are colorful, with lots of information shown in the prompt. You'll see Git status, current directory, and RVM gemset or Ruby version. Many developers replace the standard Bash shell with the Z shell and Oh-my-zsh. You don't have to install the Z shell to get a fancy prompt; the Bash-it utility is easy to install and gives you much of the functionality. A fancy prompt is helpful but requires some Unix skills to install. Don't worry about getting fancy now; you can try it down the road.

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.

Mac OS X

See this article for installation instructions:

Install Ruby on Rails – Mac OS X

Ubuntu Linux

See this article for installation instructions:

Install Ruby on Rails – Ubuntu

Hosted Computing

Nitrous.io is a browser-based development environment. Nitrous.io 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 Nitrous.io 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 installation instructions:

Install Ruby on Rails – Nitrous.io

Windows

Here are your choices for Windows:

- Use the Nitrous.io hosted development environment
- Install the Railsbridge Virtual Machine or rails-dev-box
- Use RailsInstaller for Windows as documented in Installing Rails on Windows

Nitrous.io is ideal if you have a fast Internet connection. If not, download the Railsbridge Virtual Machine or rails-dev-box to create a virtual Linux computer with Ruby 2.0 and Rails

4.0 using Vagrant. The last option, RailsInstaller, is not recommended because it does not provide an up-to-date version of Ruby or Rails. Also, RVM does not run on Windows. If you use RailsInstaller, you can still follow the tutorial; just skip the instructions that refer to RVM (though it is better to use Nitrous.io or a Vagrant virtual machine).

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 4.0)
- The third number denotes a *patch level* (Rails 4.0.1)

A major release includes new features, including changes which break backward compatibility. That means switching from Rails 3.2 to Rails 4.0 will require a significant rewrite of a Rails 3.2 application.

A minor release introduces new features but doesn't require a rewrite of the application.

A patch release fixes bugs but doesn't introduce significant features.

Ruby 2.0 and Rails 4.0

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

- The Ruby language (version 2.0.0 or newer)
- The Rails gem (version 4.0 or newer)

Open your terminal application and enter:

```
$ ruby -v
```

You might see:

```
ruby ruby-2.0.0-p247 (...)
```

You've got Ruby version 2.0.0, patch level "p247" (Ruby versions add an extra patch level to semantic versioning). Newer minor releases or patch levels are good and this tutorial will remain compatible.

Try:

```
$ rails -v
```

You might see:

```
Rails 4.0.0
```

Versions such as `4.0.0.beta1` or `4.0.0.rc2` are beta versions or "release candidates."

If you find you've installed Rails 4.0.1 or newer (a patch release), that's good. It means minor bugs have been fixed since this was written. If you find you have Rails 4.1.0 or newer (a minor release), check for a newer version of this tutorial. Minor features may have changed.

You can check for the current version of Rails here.

If you are running older versions of Ruby or Rails on your computer, you must install newer versions to avoid unexpected problems.

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 `Rails4.0.0`.

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, rbenv, or others. 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-2.0.0@learn-rails --create
$ gem install rails
```

The newest Rails version will be installed.

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 4.0.0" (or something similar).

Chapter 8

Create the Application

In previous chapters you've gained a conceptual background. In this 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 fulltime 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.

A Note About the PDF Version

This book is available in several formats, including online and PDF versions. If you are reading the PDF version on Mac OS X using the Preview application, you may find that line breaks are lost when you copy the code examples. Copying without line breaks will cause code errors. If you use Adobe Acrobat you'll be able to copy the line breaks (though indenting is lost).

If you have access to the online edition of the book you'll be able to copy and paste the code without any problem.

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 (Twitter Bootstrap). 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.

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

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.

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.

If you haven't done so already, make a folder to contain your programming projects:

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

If you are using Nitrous.io, you already have a **workspace/** folder.

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

Use Your RVM Gemset

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

```
$ rvm use ruby-2.0.0@learn-rails
$ rvm gemset list
```

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

"Rails New" Help

It's important to know that help messages are available for Rails commands.

Rails provides the `rails new` command to create a basic Rails application.

Let's see the help for the `rails new` command. Type:

```
$ rails new --help
```

You'll see a help message that shows a list of command options and an explanation of what the "rails new" command will do. The help message may seem a bit cryptic but it is worth reading to see what options are available. We won't use any of the options; what's important to us is that the "rails new" command creates a directory (project folder) and generates subfolders and files within it.

Use "Rails New" to Build the Application

To create the Rails default starter application, type:

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

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

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 nine folders and 53 files.

It will install 44 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.

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.

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-2.0.0@learn-rails --ruby-version
```

If you see "ERROR: Gemset 'learn-rails' does not exist", perhaps you overlooked an earlier step in the *Project-Specific Gemset* section (in the previous chapter) where we created the learn-rails gemset. No matter, you can create it now:

```
$ rvm use ruby-2.0.0@learn-rails --create --ruby-version
$ gem install rails
```

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-2.0.0@learn-rails` every time you start work on your project after closing the console.

You can confirm you've created the two hidden files:

```
$ ls -1pa
./
../
.gitignore
.ruby-gemset
.ruby-version
Gemfile
Gemfile.lock
README.rdoc
Rakefile
app/
bin/
config/
config.ru
db/
lib/
log/
public/
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

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
```

You'll see:

```
=> Booting WEBrick
=> Rails 4.0.0 application starting in development on http://0.0.0.0:3000
=> Run `rails server -h` for more startup options
=> Ctrl-C to shutdown server
[...] INFO  WEBrick 1.3.1
[...] INFO  ruby 2.0.0 (2013-02-24) [x86_64-darwin12.2.0]
[...] INFO  WEBrick::HTTPServer#start: pid=38534 port=3000
```

The `rails server` command launches the default WEBrick web server that is provided with Ruby.

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.

Watching 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 WEBrick 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.

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."

Stopping the Web Server

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

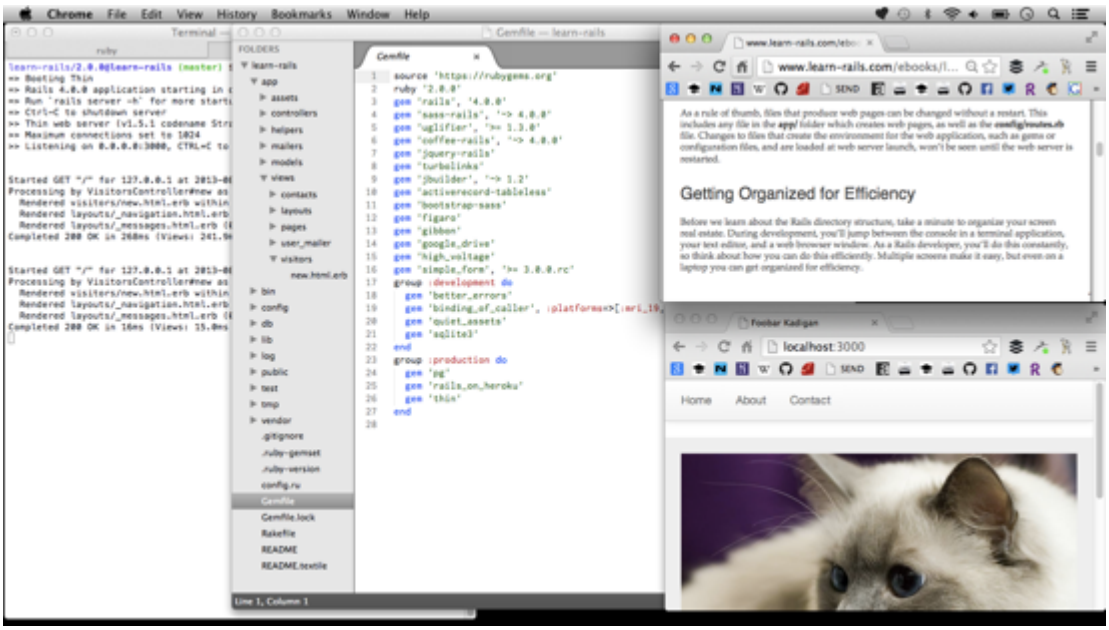
Most of the time you'll keep the web server running as you add or edit files in your project. Changes will automatically appear when you refresh the browser or request a new page. There is a tricky exception, however. If you make changes to the Gemfile, or changes to

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.

Getting 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 Sublime Text, you can open two editor panels side-by-side. Some developers find it helpful to leave the file browser 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 9

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.

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.rdoc
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 .
```

You'll learn more about each file and folder as you proceed through the tutorial. To get you started, here are two 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.

Important Folders and Files

These folders and files are important to beginners.

Gemfile	Lists all the gems used by the application.
Gemfile.lock	Lists gem versions and dependencies.
README**.rdoc**	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 important to Rails but not important to beginners.

Rakefile	Directives 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.

The App Directory

Take time to drill down into the **app/** folder in the project directory. This is easiest using the file browser. You can also use your text editor. Or you can do it with Unix commands:

```
$ cd app
$ ls -1p
assets/
controllers/
helpers/
mailers/
models/
views/
$ cd ..
```

We enter the app folder, list the contents, and then use the `cd ..` command (change directory dot dot) to return to the project directory.

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 six 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**
- **controllers**
- **helpers**
- **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.

Two folders, **mailers/** and **helpers/**, 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.

As a Rails developer, you'll spend most of your time navigating this heirarchy 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 10

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 importantly, 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. Mac OS X 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.

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.

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

# Ignore bundler config.
/.bundle

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

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

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.rdoc
# Rakefile
# app/
# bin/
# config.ru
# config/
# db/
# lib/
# log/
# public/
# vendor/
nothing added to commit but untracked files present (use "git add" to track)
```

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

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.

Committing

Staging gives you an opportunity to organize your changes in groups before you commit. If you've only worked on one feature, you'll likely stage and commit everything at once.

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. 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 repo.

Running `git status` will tell you "nothing to commit (working directory clean)."

Git Log

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

```
$ git log
```

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
```

The listing is easier to review when it is displayed in a compact format.

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. It's also important when you access your project from more than one computer or you want an offsite backup of your work.

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:

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

Now you can view your project repository on GitHub at:

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

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**.rdoc** 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.

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 the RailsApps project!"

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

```
Learning Rails
==

Learning Rails with a tutorial from the RailsApps project.
```

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

- README.rdoc uses the rdoc syntax
- README.md uses the GitHub Flavored Markdown syntax
- README.textile uses the Textile syntax

We'll use Markdown syntax by adding the `==` characters after the first line of text to force a headline.

There's no requirement that you use Markdown syntax in your README file. Markdown is a popular way to add formatting to improve readability. For us, changing the file to Markdown creates a practical exercise in using Git.

We'll use the `git mv` command to rename the file to README**.md** and save it.

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

Use `git status` to see what has changed:

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# renamed:    README.rdoc -> README.md
#
```

You could also use the Unix `mv` command to rename the file. If you do so, `git status` will show the README**.rdoc** file has been deleted and a new, untracked README**.md** file has been created.

Here's our typical workflow. We'll stage, commit, and push the change to GitHub:

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

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
```

You can read more about Git and Rails if you need more information about working with Git and GitHub for code source control.

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

Chapter 11

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.

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 watching) 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:

- actionmailer – framework for email delivery and testing
- actionpack – framework for routing and responding to web requests
- activerecord – framework for connections to databases
- activeresource – framework for manipulating data
- activesupport – utility classes and Ruby library extensions
- bundler – utility to manage gems
- railties – console commands and generators

These are the "runtime dependencies" for Rails. Each of these gems has its own dependencies as well. When you install Rails, a total of 44 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
- 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?

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. If you're a curious person, though, you might wonder where the gems live.

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

```
$ gem which bundler
/Users/me/.rvm/gems/ruby-2.0.0-p0@global/gems/bundler-1.3.5/lib/bundler.rb
$ gem which rails
/Users/me/.rvm/gems/ruby-2.0.0-p0@learn-rails/gems/railties-4.0.0/lib/rails.rb
```

Gems are files saved to the computer's disk storage. If you use RVM, they 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.

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', '4.0.0'` is "absolute": only version 4.0.0 will be used
- `gem 'rails', '>= 4.0.0'` is "optimistic": any version newer than 4.0.0 will be used
- `gem 'rails', '~> 4.0.0'` is "pessimistic"

"Pessimistic" versioning needs some explanation. `~> 4.0.0` means use any version greater than 4.0.0 and less than 4.1 (any patch version can be used). `~> 4.0` means use any version greater than 4.0 and less than 5.0 (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. The exception is the Rails gem itself. We always specify the version of Rails we are using for development.

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:

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

# Bundle edge Rails instead: gem 'rails', github: 'rails/rails'
gem 'rails', '4.0.0'

# Use sqlite3 as the database for Active Record
gem 'sqlite3'

# Use SCSS for stylesheets
gem 'sass-rails', '~> 4.0.0'

# Use Uglifier as compressor for JavaScript assets
gem 'uglifier', '>= 1.3.0'

# Use CoffeeScript for .js.coffee assets and views
gem 'coffee-rails', '~> 4.0.0'

# See https://github.com/sstephenson/execjs#readme for more supported runtimes
# gem 'therubyracer', platforms: :ruby

# Use jquery as the JavaScript library
gem 'jquery-rails'

# Turbolinks makes following links in your web application faster. Read more:
https://github.com/rails/turbolinks
gem 'turbolinks'

# Build JSON APIs with ease. Read more: https://github.com/rails/jbuilder
gem 'jbuilder', '~> 1.2'

group :doc do
  # bundle exec rake doc:rails generates the API under doc/api.
  gem 'sdoc', require: false
end

# Use ActiveModel has_secure_password
# gem 'bcrypt-ruby', '~> 3.0.0'

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

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

# Use debugger
# gem 'debugger', group: [:development, :test]
```

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 the version number. It's important to specify which version of Rails we are using. Rails changes frequently and newer versions may not work as we expect.

It's also 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 '2.0.0'
```

In the Gemfile you'll see the gems for a Rails default application, such as sqlite3, 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.

The `gem 'sdoc'` line is useful only when using `rake doc:rails` command to generate API documentation so we can remove 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.

If you remove the extra clutter in the **Gemfile** it will look like this:

```
source 'https://rubygems.org'
ruby '2.0.0'
gem 'rails', '4.0.0'

# Rails defaults
gem 'sqlite3'
gem 'sass-rails', '~> 4.0.0'
gem 'uglifier', '>= 1.3.0'
gem 'coffee-rails', '~> 4.0.0'
gem 'jquery-rails'
gem 'turbolinks'
gem 'jbuilder', '~> 1.2'
```

Adding Gems

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

I learned of these gems from a variety of different sources:

- Ruby Toolbox
- RailsCasts
- 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:

- activerecord-tableless – helps to use Rails without a database
- figaro – configuration framework
- gibbon – access to the MailChimp API
- google_drive – use Google Drive spreadsheets for data storage
- high_voltage – for static pages like "about"
- simple_form – forms made easy

We'll add these gems for the Zurb Foundation front-end framework:

- compass-rails – support for Zurb Foundation
- zurb-foundation – front-end framework

We'll also add utilities that make development easier:

- better_errors – helps when things go wrong
- quiet_assets – suppresses distracting messages in the log
- rails_layout – generates files for an application layout

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

```
source 'https://rubygems.org'
ruby '2.0.0'
gem 'rails', '4.0.0'

# Rails defaults
gem 'sqlite3'
gem 'sass-rails', '~> 4.0.0'
gem 'uglifier', '>= 1.3.0'
gem 'coffee-rails', '~> 4.0.0'
gem 'jquery-rails'
gem 'turbolinks'
gem 'jbuilder', '~> 1.2'

# learn-rails
gem 'activerecord-tableless'
gem 'compass-rails', '~> 2.0.alpha.0'
gem 'figaro'
gem 'gibbon'
gem 'google_drive'
gem 'high_voltage'
gem 'simple_form'
gem 'zurb-foundation'
group :development do
  gem 'better_errors'
  gem 'quiet_assets'
  gem 'rails_layout'
end
```

Notice that we've placed the better_errors and quiet_assets 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*.

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

Install the Gems

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 gem metadata from https://rubygems.org/..
Resolving dependencies...
Using rake (10.0.4)
Using i18n (0.6.4)
Installing minitest (4.7.4)
.
.
.
(many more gems not shown... you get the idea)
.
.
.
Your bundle is complete!
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.

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 12

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 save an account name and password for a Gmail account so we can send email from the application and save data to a Google Drive spreadsheet.

We'll also need to store an API key to access MailChimp, which we'll use to add visitors' email addresses to a mailing list.

Remote Git repositories such as GitHub are a place to store and share code. But you shouldn't save email account credentials or private API keys to a shared Git repository where others can see them. Operating systems (Linux, Mac OS X, Windows) provide mechanisms to set local environment variables, as does Heroku and other deployment platforms. Environment variables can be accessed from Rails applications and provide an ideal place to set configuration settings that must remain private. With a bit of Unix skill, most developers can set environment variables using the Unix shell. For applications from the RailsApps project, we take a hybrid approach and use the figaro gem that lets us set environment variables from the Unix shell or from a simple configuration file.

For our tutorial, we'll show how to set up configuration settings using the figaro gem. You can read the article Rails Environment Variables if you'd like learn about the figaro gem or explore other approaches.

Configuration File

The tutorial application uses the figaro gem to set environment variables. We've already installed the figaro gem in the Gemfile and run `bundle install`.

The figaro gem uses a generator to set up the necessary files. Run:

```
$ rails generate figaro:install
```

Rails provides the `rails generate` command to be used by gems that need to modify Rails files or install configuration files.

Using the `rails generate` command, the figaro gem generates a **config/application.yml** file and lists it in your **.gitignore** file. The **.gitignore** file prevents the **config/application.yml** file from being saved in the Git repository so your credentials are kept private.

The figaro generator will create a file with some example entries:

```
# Add application configuration variables here, as shown below.
#
# PUSHER_APP_ID: "2954"
# PUSHER_KEY: 7381a978f7dd7f9a1117
# PUSHER_SECRET: abdc3b896a0ffb85d373
# STRIPE_API_KEY: EdAvEPVEC3LuaTg5Q3z6WbDVqZlcBQ8Z
# STRIPE_PUBLIC_KEY: pk_BRgD57O8fHja9HxduJUszhef6jCyS
```

These are examples; we don't need them.

Use your text editor to replace the file **config/application.yml** with this:

```
# Add account credentials and API keys here.
# See http://railsapps.github.io/rails-environment-variables.html
# This file should be listed in .gitignore to keep your settings secret!
# Each entry sets a local environment variable and overrides ENV variables in the Unix
shell.
GMAIL_USERNAME: Your_Username
GMAIL_PASSWORD: Your_Password
MAILCHIMP_API_KEY: Your_MailChimp_API_Key
MAILCHIMP_LIST_ID: Your_List_ID
DOMAIN_NAME: example.com
OWNER_EMAIL: me@example.com
```

Next, set credentials for your Gmail and MailChimp accounts.

Replace the placeholders in the **config/application.yml** file with real credentials.

Make sure there is a space after each colon and before the value for each entry.

Here's an important note: If any value contains non-alphanumeric characters (punctuation marks), you must enclose the value in quotation marks. If you don't, you'll get an error when starting the web server.

All configuration values in the **config/application.yml** file are available anywhere in the application as environment variables. We'll use the environment variables to configure the tutorial application to send email.

Gmail

For the Gmail username and password, enter the credentials you use to log in to Gmail when you check your inbox.

If you don't have a Gmail account, you can sign up for an account with Mandrill instead and follow instructions in the article Send Email with Rails.

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`. Log in to MailChimp to get your API key. Click your name at the top of the navigation menu, then click "Account Settings." Click "Extras," then "API keys." You have to generate an API key; MailChimp doesn't create one automatically.

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.

Domain Name

If you already have a custom domain name you'll use when you deploy the application, you can replace `example.com` now. If not, leave `example.com` in place for now. Later, if you follow the tutorial to deploy the application to Heroku, you'll replace `example.com` with the unique name you've given your application on Heroku. You'll have to wait until you deploy to know the name you'll use on Heroku.

We'll use the domain name variable when we configure email for delivery in production.

Owner Email

You'll send email messages to this address when a visitor submits a contact request form. Replace `me@example.com` with an email address where you receive mail.

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 adminstrator 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. For convenience during development, you can use your own Gmail account. In production, for high volume transactional email and improved deliverability, it's best to use a service such as Mandrill or Mailgun. See the article Send Email with Rails.

We need to configure Rails so the application can connect with an email server. For our tutorial application, we'll connect to Gmail to send email.

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.gmail.com",
  port: 587,
  domain: ENV["DOMAIN_NAME"],
  authentication: "plain",
  enable_starttls_auto: true,
  user_name: ENV["GMAIL_USERNAME"],
  password: ENV["GMAIL_PASSWORD"]
}
```

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 the `ENV["GMAIL_USERNAME"]` and `ENV["GMAIL_PASSWORD"]` environment variables that we set in the **config/application.yml** file. We could "hard code" a username and password here but that would expose confidential data if your GitHub repository is

public. Using environment variables from the **config/application.yml** file keeps your secrets safe.

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

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 13

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.

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
```

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:

```
LearnRails::Application.routes.draw do
  root to: redirect('/about.html')
end
```

Notice the name of the application `LearnRails` is included in the file. When you created the application, I suggested you use the name `learn-rails`. If you gave the project a different name, you'll have to modify the **config/routes.rb** file accordingly.

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.

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 14

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

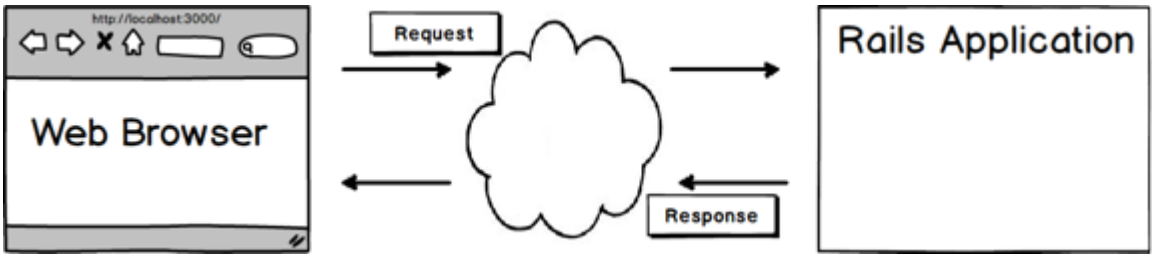
Remember, at its core, the World Wide Web is nothing more than web browsers that request files from web servers.

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 there is nothing but 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.



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. Some developers prefer Mozilla Firefox and Apple Safari but Google Chrome is the most popular. Even if you prefer one of the others, 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. 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.

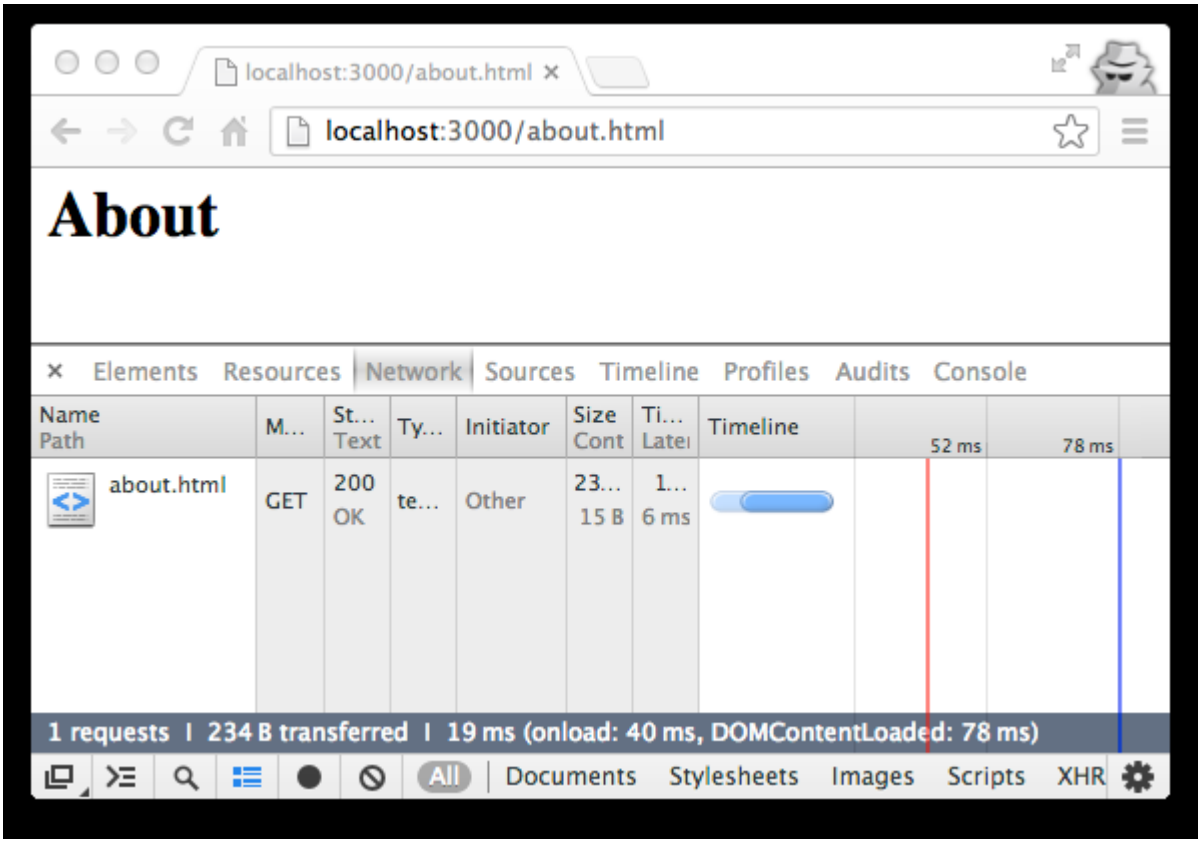
In Chrome on Mac OS X, press Command-Option-I to open the *Developer Tools View* in a section of the browser window. Alternatively, you can find the menu item under View/Developer/Developer Tools.

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

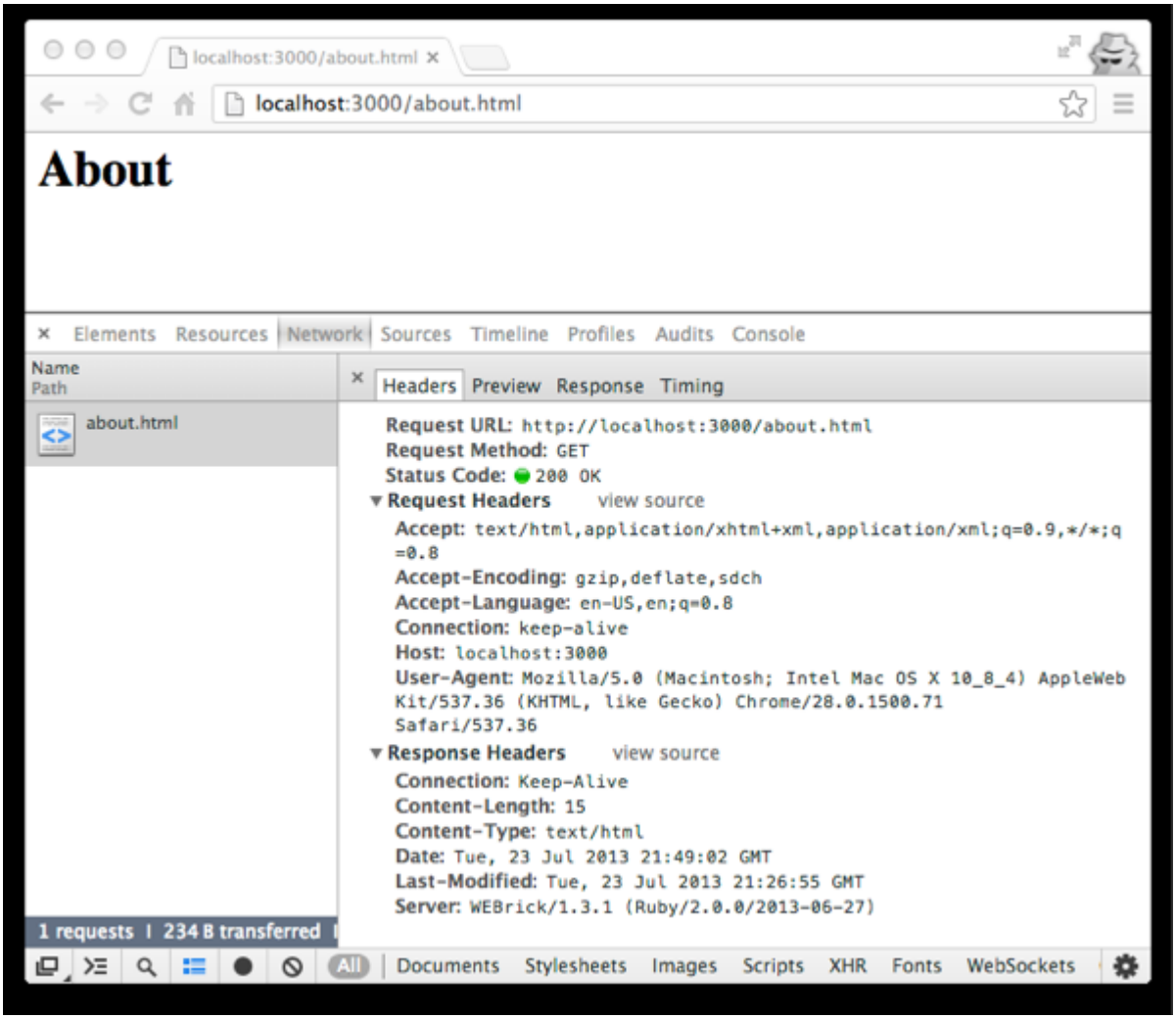
Select the Network tab in the Developer Tools View.

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 identifer)

The response is composed of

- status code (200 OK or 304 Not Modified)
- response headers (including date/time and server identifier)
- 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 127.0.0.1 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." The browser's "Request URL:http://localhost:3000/" matches the server's "'/' for 127.0.0.1" (localhost is at IP address 127.0.0.1).

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.

Document Object Model

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. When all the files are present and processed, the browser fires a "DOM ready" event.

The Document Object Model (DOM) is an API for HTML documents. It provides a structural representation of the document, enabling you to modify its content and visual presentation by using a scripting language such as JavaScript.

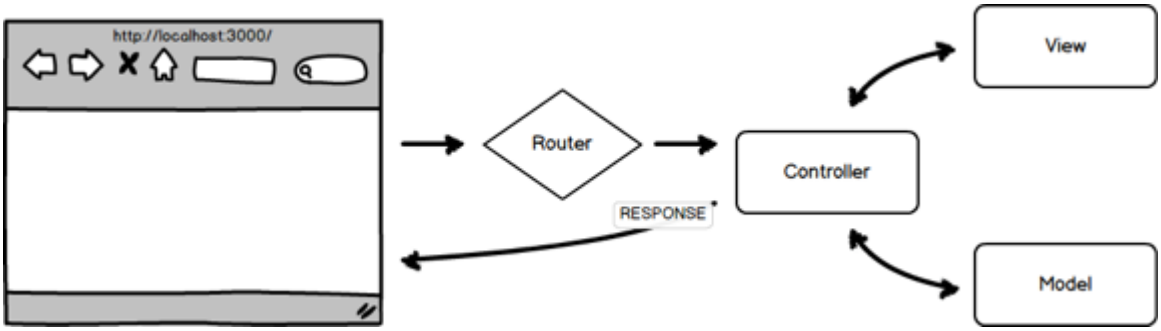
"DOM ready" is the starting gun for any interactive features of the web page, such as drop-down menus or graphics carousels.

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.

Model View Controller

Now that we've investigated the request-response cycle, let's dig deeper to understand what happens inside the Rails application server in response to a browser request.

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



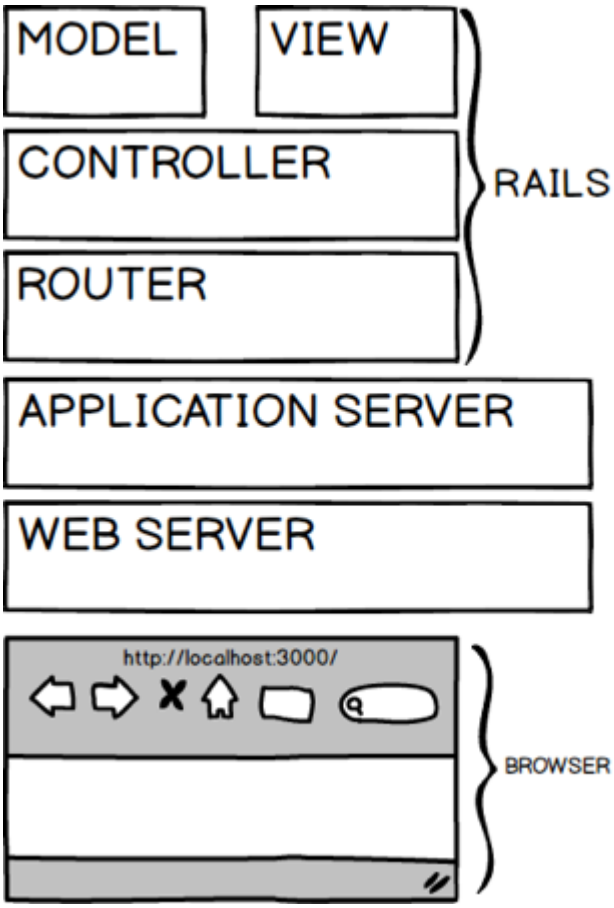
You learned earlier that, from the perspective of a software architect, Rails is organized to conform to the model–view–controller software design pattern. This enforces "separation of concerns" to keep code manageable and organized. The MVC design pattern is optimal for web applications and is a central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

Now we'll look at ways to implement the home page using the full power of Rails.

Chapter 15

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.

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

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.

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 < ActiveRecord::Base` – the model class name is capitalized and singular
- `class VisitorsController < ApplicationController` – for a controller, combine a pluralized model name with "Controller" in CamelCase

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:

```
LearnRails::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.

Notice that the name of the application is contained in the **config/routes.rb** file. Earlier, I recommended using "learn-rails" as the name of the application so you will not need to change the code here.

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.

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**:

```
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, 9, 22)` to the `birthdate` variable.

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**:

```
<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. This directory is for any views associated with the Visitors controller. We have only a single `new` view but if we had more views, 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.

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

```
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:

```
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.

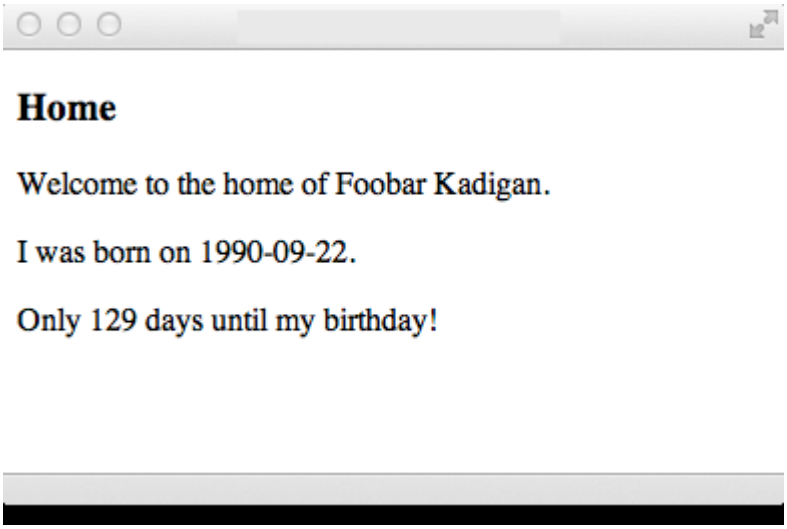
Test the Application

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.



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 stucture 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 16

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
2.0.0p0 :001 >
```

The command `irb` launches the program and displays a prompt that show your Ruby version, a line number, and an arrow. If you enter a valid Ruby expression, the interpreter will display the result of evaluating the expression.

Try simple arithmetic:

```
2.0.0p0 :001 > n = 2
 => 2
2.0.0p0 :002 > 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:

```
2.0.0p0 :001 > n = 10
 => 10
2.0.0p0 :002 > if n < 10
2.0.0p0 :003?>   puts "small"
2.0.0p0 :004?>   else
2.0.0p0 :005 >     puts "big"
2.0.0p0 :006?>   end
big
 => nil
2.0.0p0 :007 >
```

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:

```
2.0.0p0 :001 > 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
2.0.0p0 :001 > exit
```

Learn More About IRB

Here's an entertaining way to learn about IRB:

- Why's (Poignant) Guide to Ruby

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 4.0.0)
2.0.0p0 :001 >
```

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:

```
2.0.0p0 :001 > myboss = Owner.new
 => #<Owner:0x007fe885163aa8>
```

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 an object 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 instantiated from its class.

Let's assign the name of our boss to a variable called `name`:

```
2.0.0p0 :002 > 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*:

```
2.0.0p0 :003 > 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 when you visit the application home page:

```
Started GET "/" for 127.0.0.1 at ...
Processing by VisitorsController#new as HTML
  Rendered visitors/new.html.erb within layouts/application (48.8ms)
Completed 200 OK in 233ms (Views: 211.5ms | ActiveRecord: 0.0ms)
```

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 127.0.0.1 at ...
Processing by VisitorsController#new as HTML
DEBUG: entering new method
DEBUG: Owner name is Foobar Kadigan
  Rendered visitors/new.html.erb within layouts/application (0.2ms)
Completed 200 OK in 8ms (Views: 4.6ms | 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**:

```
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 127.0.0.1 at ...
Processing by VisitorsController#new as HTML
DEBUG: entering new method
DEBUG: Owner name is Foobar Kadigan
DEBUG: entering Owner countdown method
  Rendered visitors/new.html.erb within layouts/application (0.3ms)
Completed 200 OK in 7ms (Views: 4.2ms | 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 127.0.0.1 at ...
Processing by VisitorsController#new as HTML
DEBUG: entering new method
DEBUG: Owner name is Foobar Kadigan
DEBUG: entering Owner countdown method
  Rendered visitors/new.html.erb within layouts/application (0.3ms)
Completed 200 OK in 5ms (Views: 4.2ms | 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 "Request URL:http://localhost:3000/" matches the server's "'/' for 127.0.0.1" (localhost is at IP address 127.0.0.1). 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.

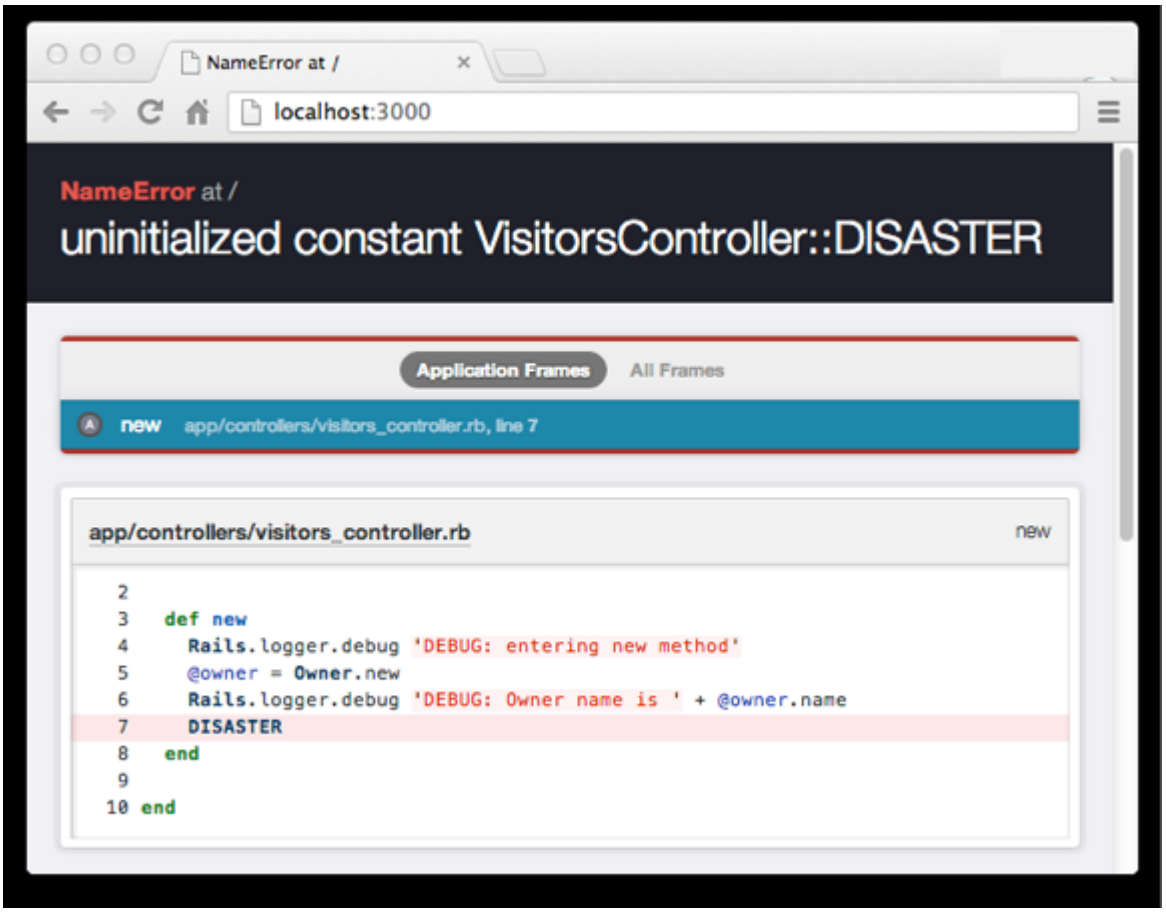
Modify the file **app/controllers/visitors_controller.rb**:

```
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 127.0.0.1 at ...
Processing by VisitorsController#new as HTML
DEBUG: entering new method
DEBUG: Owner name is Foobar Kadigan
Completed 500 Internal Server Error in 10ms

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 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 ("barfed") when it encountered an "uninitialized constant" at line 7 of the **app/controllers/visitors_controller.rb** file in the `index` 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 developers 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**:

```
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 127.0.0.1 at ...
Processing by VisitorsController#new as HTML
DEBUG: entering new method
DEBUG: Owner name is Foobar Kadigan
Completed 500 Internal Server Error in 22ms

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/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
```

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 <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." 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 17

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**:

```
class Example < Object

  # This is a comment.

  attr_accessor :honorific
  attr_accessor :name
  attr_accessor :date

  def to_s
    @name
  end

  def initialize(name,date)
    @name = name
    @date = date.nil? ? Date.today : date
  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.

We know that Foobar Kadigan was born in December. Perhaps the code could be used to display a list of famous people who were also born in December.

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

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 that class. A given house is an *instance*.

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 the concept "vehicle." The blue car in your driveway with four doors is an *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.

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:

abstraction
: a concept that has a relationship to other concepts

class
: an abstraction that encapsulates data and behavior

class definition
: written code that describes a class

instance or **object**
: a unique version 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 way to get a result 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.

Ruby Files

When we write code, we save it in files. Rails provides a directory structure with "assigned parking" so that Ruby files are automatically loaded when you start the web server. Our miscellaneous example file goes in 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
2.0.0p0 :001 > load 'lib/example.rb'
 => true
2.0.0p0 :002 > require 'date'
 => true
2.0.0p0 :003 > ex = Example.new('Daniel',nil)
 => #<Example:... @name="Daniel", @date=#<Date: 2013-09-03 ((...)>>
2.0.0p0 :004 > 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 of names.

Remember you can use Control-d to exit from IRB.

Now, for practice, we'll read the Ruby code.

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.

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.

Attributes

In Ruby, attributes are also called properties.

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.

Here we use the `attr_accessor` directive to specify that we want to enable access to `honorific`, `name` and `date` attributes.

```
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:

```
ex = Example.new('Daniel',nil)
my_name = ex.name
```

In Ruby, attributes or properties are just a specialized form of 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 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:

```
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.

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.

Here we define a method `initialize` that takes `name` and `date` arguments.

```
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.

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.

```
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.

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 inside a method 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.

Instance Variable

An ordinary variable only retains its assigned value within its most immediate surroundings. If you assign a variable inside a method, the variable only can be used inside the method.

Often you want a variable to be useful throughout a class, in 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
```

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.

Remember the story of the six blind men and the elephant? 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:

```
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:

```
condition ? value_if_true : value_if_false
```

Here is the ternary operator we use in our example code:

```
@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:

```
name = 'Foobar Kadigan'
```

We can also perform "string addition" to concatenate strings. Here we add an honorific, a space, and a name:

```
@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:

```
@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 definition. This is common when you need a simple utility method that is used by several other methods.

Use the keyword `private` to indicate the method should not be accessed by a call to the object from outside the instance. Any methods that follow the keyword `private` are only used by other methods in the same object.

```
private
```

You often see private methods in Rails. Ruby provides a *protected* keyword as well, but it is seldom seen in Rails applications. *Protected* methods can be invoked only by objects of the defining class and its subclasses.

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:

```
born_in_december = [ ]
```

We can populate the array with values when we create it:

```
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:

```
my_list = %w( apples oranges )
```

We can add new elements to an array with a `push` method:

```
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
```

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.

Online

- TryRuby.org – free browser-based interactive tutorial from Code School
- Codecademy Ruby Track – free browser-based interactive tutorials from Codecademy
- Ruby Monk – free browser-based interactive tutorial from C42 Engineering
- 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

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

Newsletters

- Practicing Ruby – $8/month for access to over 90 helpful articles on Ruby

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
```

Or 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 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 18

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.

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>
<link data-turbolinks-track="true" href="/assets/application.css?body=1" media="all"
rel="stylesheet" />
<script data-turbolinks-track="true" src="/assets/jquery.js?body=1"></script>
<script data-turbolinks-track="true" src="/assets/jquery_ujs.js?body=1"></script>
<script data-turbolinks-track="true" src="/assets/turbolinks.js?body=1"></script>
<script data-turbolinks-track="true" src="/assets/application.js?body=1"></script>
<meta content="authenticity_token" name="csrf-param" />
<meta content="NRPrgfuj5GAyylNpNxQaMHDypcOsu6dmh5DT1yET6hQ=" name="csrf-token" />
</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.

Let's examine the application layout file.

Open the file **app/views/layouts/application.html.erb**:

```
<!DOCTYPE html>
<html>
<head>
  <title>LearnRails</title>
  <%= stylesheet_link_tag "application", media: "all", "data-turbolinks-track" => true %>
  <%= javascript_include_tag "application", "data-turbolinks-track" => true %>
  <%= csrf_meta_tags %>
</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 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.erb**:

```
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`.

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

You can 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:

```
<!DOCTYPE html>
<html>
<head>
  <title>LearnRails</title>
  <%= stylesheet_link_tag "application", media: "all", "data-turbolinks-track" => true %>
  <%= javascript_include_tag "application", "data-turbolinks-track" => true %>
  <%= csrf_meta_tags %>
</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:

```
<% content_for :sidebar do %>
  <h3>Contact Info</h3>
  <p>Email: me@example.com</p>
<% end %>
<h3>Main</h3>
<p>Welcome!</p>
```

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

Let's look closely at the application layout file.

As we might expect, we see HTML tags, enclosed in angle brackets (sometimes called chevrons).

We also see ERB `<%= ... %>` delimiters. In the application layout file, the `<%= ... %>` delimiters don't include anything 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 programming language was famous for its mantra "Code should read like a conversation." 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 content="authenticity_token" name="csrf-param" />
<meta content="NRPrgfuj5GAyylNpNxQaMHDypcOsu6dmh5DT1yET6hQ=" name="csrf-token" />
```

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 RailsLayout 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 Zurb Foundation front-end framework.

To make it easy, we'll use the RailsLayout 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 Zurb Foundation.

In your **Gemfile**, you've already added:

```
gem 'rails_layout'
```

and previously run `$ bundle install`.

You previously used the `rails generate` command to set up configuration files with the Figaro gem. Any gem that needs default files can use the `rails generate` command to run a simple script that creates files.

The RailsLayout gem uses the `rails generate` command to set up files we need. Run:

```
$ rails generate layout simple --force
```

The `--force` argument will force the gem to replace the existing **app/views/layouts/application.html.erb** file.

The gem will add four files to your project:

- **app/views/layouts/application.html.erb**
- **app/views/layouts/_messages.html.erb**
- **app/views/layouts/_navigation.html.erb**
- **app/assets/stylesheets/simple.css**

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 four 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" => true
%>
    <%= javascript_include_tag "application", "data-turbolinks-track" => true %>
    <%= csrf_meta_tags %>
  </head>
  <body>
    <div id="container">
      <header>
        <%= render 'layouts/navigation' %>
      </header>
      <main role="main">
        <%= render 'layouts/messages' %>
        <%= yield %>
      </main>
      <footer>
      </footer>
    </div>
  </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 Twitter Bootstrap or Zurb Foundation.

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 RailsLayout 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:

```
<!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:

```
<!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" => true %>
<%= javascript_include_tag "application", "data-turbolinks-track" => true %>
```

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 you'll automatically get one single file delivered to the browser.

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.

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.html.erb**:

```
<%= render 'layouts/footer' %>
```

Notice that you specify the folder within the **app/views/** directory with a truncated 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.com/about.html`. Absolute URLs are a headache when moving the site, for example from `staging.example.com` 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 broken links. Instead, with the Rails `link_to` helper, you always get the destination location specified in the **config/routes.rb** file.

Navigation Partial

Open the file **app/views/layouts/_navigation.html.erb**:

```
<ul class="nav">
  <li><%= link_to 'Home', root_path %></li>
</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.

We use the navigation links partial in our application layout with the expression:

```
<%= render 'layouts/navigation' %>
```

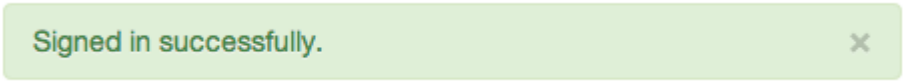
Examine the **app/views/layouts/application.html.erb** and you'll see the use of the navigation links partial.

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:



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:

```
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_controller.rb**:

```
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:

```
flash.each do |key, value|
  puts '<div class="' + key + '">' + value + '</div>'
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.

HTML5 Elements

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 *container div*:

```
<div id="container">
```

This is a common technique among front-end designers, particularly useful for adding margins or borders to the page layout.

You'll also notice tags that are structural elements in the HTML5 specification:

- `<header>`
- `<main>`
- `<footer>`

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.

Finally, the `<footer>` tag 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 one between the `<footer>` tags, if you want.

Application Layout

Let’s look again at the **app/views/layouts/application.html.erb** file.

We don’t have to add anything because the RailsLayout gem has created everything we need.

```
<!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" => true
%>
    <%= javascript_include_tag "application", "data-turbolinks-track" => true %>
    <%= csrf_meta_tags %>
  </head>
  <body>
    <div id="container">
      <header>
        <%= render 'layouts/navigation' %>
      </header>
      <main role="main">
        <%= render 'layouts/messages' %>
        <%= yield %>
      </main>
      <footer>
      </footer>
    </div>
  </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.

Our application layout is complete—for now. In the next chapter, we'll revise it to support styling with Zurb Foundation.

Simple CSS

So far, we've examined three files that were added by the RailsLayout gem:

- **app/views/layouts/application.html.erb**
- **app/views/layouts/_messages.html.erb**
- **app/views/layouts/_navigation.html.erb**

Let's examine the CSS file that was created by the RailsLayout gem.

Open the file **app/assets/stylesheets/simple.css**:

```
/*
 * Simple CSS stylesheet for a navigation bar and flash messages.
 */
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 been given 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 Zurb Foundation to supply styles for the header, navigation links, and flash messages.

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 19

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 Zurb Foundation 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

As a programming language, JavaScript is as powerful as Ruby or any other established language. Unfortunately, JavaScript doesn't include a native package manager for code libraries. There is no JavaScript version of RubyGems. Despite this obstacle, availability of open source JavaScript libraries has grown in recent years.

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. None of these JavaScript frameworks dominate web application development like Ruby on Rails, but they are quickly gaining popularity and are said to represent the future of web development. We won't look at Ember.js, AngularJS, or Backbone.js in this book; they are an advanced topic and require entire books themselves.

Front-End Frameworks

Twitter Bootstrap and Zurb Foundation are front-end frameworks, a hybrid of 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.

Twitter Bootstrap is the best-known front-end framework. It is 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.

Zurb Foundation was released to as an open source project in October 2011, after more than a year of internal use at Zurb, a Silicon Valley design consultancy.

Both Twitter Bootstrap and Zurb Foundation are popular among Rails developers. They each use jQuery and add a library of standardized CSS for the most commonly needed web page features.

Just ahead, we'll look at why we use Zurb Foundation in this book. But first, you'll need to learn about LESS and Sass.

CSS Preprocessing with LESS or Sass

Ordinary CSS is not a programming language. As a result, CSS rules are verbose and often repetitive. To add efficiency to CSS, Twitter Bootstrap and Zurb Foundation rely on CSS preprocessors; LESS for Twitter Bootstrap and the Sass project for Zurb Foundation. LESS and Sass extend 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 generally recognized as more powerful than LESS, and Sass is included in any new Rails application. The creators of Twitter Bootstrap recently explained, Why Less?, and it seems a primary reason was their greater comfort with JavaScript (the underlying language of the LESS preprocessor) than with Ruby (the language underlying Sass).

Twitter Bootstrap or Zurb Foundation?

Which should you use, Twitter Bootstrap or Zurb Foundation?

Twitter Bootstrap has a larger developer community and 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.

However, Zurb Foundation is gaining popularity with Rails developers. One factor is direct support for Rails by the creators of Foundation. Zurb provides a gem that installs Foundation in any Rails application. When Zurb releases new versions of Foundation, the company updates the gem themselves. Foundation's use of Sass means easier integration with Rails applications.

There are several versions of Twitter Bootstrap that have been converted to a Sass implementation but they tend to lag behind Twitter Bootstrap releases because they are not directly supported by the Bootstrap creators. For example, at the time this book was written, Twitter Bootstrap 3.0 had been released but gems for the new version were not yet out.

Foundation's use of Sass, and Zurb's direct support of a Foundation gem for Rails, are among the reasons that Foundation is gaining popularity in the Rails community.

If you're eager to try Twitter Bootstrap, the RailsApps project provides a Rails Bootstrap example application and an accompanying tutorial. Learning about Twitter Bootstrap is a great way to expand your knowledge as a next step after you complete this book.

Before I show you how to integrate Zurb Foundation 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 Twitter 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 Twitter Bootstrap or Zurb Foundation 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 Twitter Bootstrap or Zurb Foundation 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 Twitter Bootstrap or Zurb Foundation than to leak ugliness onto the web.

A third option is to purchase 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 WordPress, Tumblr, or CMS applications such as Drupal or Joomla. They don't offer themes for Rails and it is not easy to adapt one of their themes for a Rails application. I'm only aware of one firm that sells prepackaged themes for Rails applications using Zurb Foundation: RailsThemes (it is worth a look to see what can be done). An alternative is to convert open source themes designed with Twitter Bootstrap, such as themes from Start Bootstrap, Bootswatch, or the Themestrap gallery.

Even if you use a prepackaged theme, you'll need to know how to set up a front-end framework in Rails. We'll look at setting up Zurb Foundation next.

Zurb Foundation Gem

Zurb Foundation 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. Zurb Foundation is packaged as a gem.

In your **Gemfile**, you've already added:

```
gem 'compass-rails', '~> 2.0.alpha.0'
gem 'zurb-foundation'
```

and previously run `$ bundle install`.

Zurb Foundation requires the compass-rails gem which (at the time this was written) is available in a prerelease version for Rails 4.0.

Rather than following the installation instructions provided in the Foundation 4 Documentation, we'll use the RailsLayout gem to set up Zurb Foundation and create the files we need. Our approach is slightly different from the Zurb instructions but yields the same results.

RailsLayout Gem with Zurb Foundation

In the previous chapter, we used the RailsLayout gem to configure the default application layout with HTML5 elements, navigation links, and flash messages. Now we'll use the RailsLayout gem to set up Zurb Foundation 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 RailsLayout gem uses to set up Foundation and add the necessary files. Run:

```
$ rails generate layout foundation4 --force
```

With the `--force` argument, the RailsLayout gem will replace existing files.

The RailsLayout gem will rename the file:

- **app/assets/stylesheets/application.css**

to:

- **app/assets/stylesheets/application.css.scss**

It will create the file:

- **app/assets/stylesheets/foundation_and_overrides.css.scss**

and modify the file:

- **app/assets/javascripts/application.js**

The gem will replace three files:

- **app/views/layouts/application.html.erb**
- **app/views/layouts/_messages.html.erb**
- **app/views/layouts/_navigation.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 Zurb Foundation.

Renaming the application.css File

The RailsLayout 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 RailsLayout gem renamed the **app/assets/stylesheets/application.css** 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 files
 * listed below.
 *
 * Any CSS and SCSS file within this directory, lib/assets/stylesheets, vendor/assets/
stylesheets,
 * or vendor/assets/stylesheets of plugins, if any, can be referenced here using a
relative path.
 *
 * You're free to add application-wide styles to this file and they'll appear at the
top of the
 * compiled file, but it's generally better to create a new file per style scope.
 *
 *= require_self
 *= require_tree .
 */
```

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.

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 Zurb Foundation. These modifications don't belong in the **app/assets/stylesheets/application.css.scss** file; they will go in the **app/assets/stylesheets/foundation_and_overrides.css.scss** file.

In general, it's bad practice to place a lot of CSS in the **app/assets/stylesheets/application.css.scss** 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 comments 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 generated 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 administrative 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.

CSS Example

The RailsLayout gem modifies the file **app/assets/stylesheets/application.css.scss** to add a nice gray box as a background to page content. This gives us an example of adding a CSS

rule that will be used on every page of the application. It is independent of Zurb Foundation so it can be placed in the **app/assets/stylesheets/application.css.scss** file.

Take a look at this excerpt from the **app/assets/stylesheets/application.css.scss** file:

```
.content {
  background-color: #eee;
  padding: 20px;
  margin: 0 -20px;
  -webkit-border-radius: 0 0 6px 6px;
  -moz-border-radius: 0 0 6px 6px;
  border-radius: 0 0 6px 6px;
  -webkit-box-shadow: 0 1px 2px rgba(0,0,0,.15);
  -moz-box-shadow: 0 1px 2px rgba(0,0,0,.15);
  box-shadow: 0 1px 2px rgba(0,0,0,.15);
}
```

The CSS code applies styling to a `.content` class. It sets background color, a border and shadow, padding and margin. It's complicated by accommodating differences among web browsers.

Zurb Foundation CSS

The RailsLayout gem added a file **app/assets/stylesheets/foundation_and_overrides.css.scss** containing:

```
// Required global settings and mixins for Foundation
@import "foundation";
```

The file **app/assets/stylesheets/foundation_and_overrides.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 `@import "foundation";` directive will import the Foundation CSS rules from the Foundation gem.

You could add the Foundation `@import` code to the **app/assets/stylesheets/application.css.scss** file. However, it is better to have a separate **app/assets/stylesheets/foundation_and_overrides.css.scss** file. You may wish to modify the Foundation CSS rules; placing changes to Foundation CSS rules in the **foundation_and_overrides.css.scss** file will keep your CSS better organized.

Zurb Foundation JavaScript

Zurb Foundation provides both CSS and JavaScript libraries.

Like the **application.css.scss** file, the **application.js** file is a manifest that allows a developer to designate the JavaScript files that will be combined for delivery to the browser.

The RailsLayout gem modified the file **app/assets/javascripts/application.js** to include the Foundation JavaScript libraries:

```
//= require jquery
//= require jquery_ujs
//= require turbolinks
//= require foundation
//= require_tree .
$(function() {
  $(document).foundation();
});
```

It added the directive `//= require foundation` before `//= require_tree .`.

The last three lines use jQuery to load the Foundation JavaScript libraries after the browser has fired a "DOM ready" event (which means the page is fully rendered and not waiting for additional files to download).

```
$(function() {
  $(document).foundation();
});
```

Note that this configuration is different from the instructions provided in the Foundation 4 Documentation. In keeping with Rails best practices, we load the Foundation JavaScript libraries using the asset pipeline in the `<head>` section of the default application layout. Using the jQuery "DOM ready" event to load Foundation insures that Foundation is compatible with other jQuery plugins or JavaScript code.

Application Layout with Zurb Foundation

Let's look at the new application layout file.

Examine the contents of 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" => true
%>
    <%# Modernizr is required for Zurb Foundation 4 %>
    <%= javascript_include_tag "vendor/custom.modernizr" %>
    <%= javascript_include_tag "application", "data-turbolinks-track" => true %>
    <%= csrf_meta_tags %>
  </head>
  <body>
    <%# application layout styled for Zurb Foundation 4 %>
    <header>
      <nav class="top-bar">
        <ul class="title-area">
          <li class="name"><!-- add site name or logo here --></li>
          <li class="toggle-topbar menu-icon"><a href="#"><span>Menu</span></a></li>
        </ul>
      <section class="top-bar-section">
        <%= render 'layouts/navigation' %>
      </section>
    </nav>
    </header>
    <main role="main">
      <div class="container">
        <div class="content">
          <div class="row">
            <div class="twelve columns">
              <%= render 'layouts/messages' %>
              <%= yield %>
            </div>
          </div>
          <footer>
          </footer>
        </div>
      </div>
    </main>
  </body>
</html>
```

Modernizr JavaScript Library

You'll see the file now includes:

```
.
.
.
<%# Modernizr is required for Zurb Foundation 4 %>
<%= javascript_include_tag "vendor/custom.modernizr" %>
.
.
.
```

The Modernizr JavaScript library is a prerequisite for Foundation. Modernizr acts as a shim for HTML5 elements for older browsers as well as detecting mobile devices. It must be loaded before Foundation, so it is included above `javascript_include_tag "application"`.

Navigation Bar

The `<header>` section is now more complex, with layout and CSS classes needed to produce a navigation bar:

```
.
.
.
<header>
  <nav class="top-bar">
    <ul class="title-area">
      <li class="name"><!-- add site name or logo here --></li>
      <li class="toggle-topbar menu-icon"><a href="#"><span>Menu</span></a></li>
    </ul>
  <section class="top-bar-section">
    <%= render 'layouts/navigation' %>
  </section>
</nav>
</header>
.
.
.
```

The complexity is required to render a responsive navigation bar. At the conclusion of this chapter, you can test it 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 Zurb Foundation 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 text `<!-- add site name or logo here -->`. 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 navigation partial `render 'layouts/navigation'` with a Foundation class to complete the navigation bar.

Main Section

The `<main>` section is now wrapped in divs and classes to use the Foundation layout grid:

```
.
.
.
 <main role="main">
   <div class="container">
     <div class="content">
       <div class="row">
         <div class="twelve columns">
           <%= render 'layouts/messages' %>
           <%= yield %>
.
.
.
```

The `container` and `content` classes are ours, not part of Foundation. The `row` and `twelve columns` classes set up the Foundation grid. We wrap both the flash messages partial `render 'layouts/messages'` and the main page content rendered by `yield` within the Foundation grid.

Navigation Partial with Zurb Foundation

Examine the file **app/views/layouts/_navigation.html.erb**:

```
<ul>
  <li><%= link_to 'Home', root_path %></li>
</ul>
```

Later we'll add links to "About" and "Contact" pages.

The navigation partial is simply a list of navigation links. It doesn't require additional CSS styling. The layout and styling required for the Foundation navigation bar are in the default application layout file.

Flash Messages with Zurb Foundation

The messages partial we use with Zurb Foundation is complex.

Examine the file **app/views/layouts/_messages.html.erb**:

```
<%# Rails flash messages styled for Zurb Foundation 4 %>
<% flash.each do |name, msg| %>
  <% if msg.is_a?(String) %>
    <div data-alert class="alert-box round <%= name == :notice ? "success" : "alert" %>">
      <%= content_tag :div, msg %>
      <a href="#" class="close">&times;</a>
    </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 Foundation CSS styling around the message. Foundation recognizes a class `alert-box` and `round` (for rounded corners). A class of either `success` or `alert` styles the message. Rails `notice` messages will get styled with the Foundation `success` class. Any other Rails messages, including `alert` messages, will get styled with the Foundation `alert` class.

We use the Rails `content_tag` view helper to create a div containing the message.

Finally, we create a "close" icon by applying the class `close` to a link. We use the HTML entity `&times;` (a big "X" character) for the link; it could be the word "close" or anything else we like. Foundation's integrated JavaScript library will hide the alert box when the "close" link is clicked.

Foundation provides detailed documentation if you want to change the styling of the alert boxes.

Set up SimpleForm with Zurb Foundation

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 SimpleForm helpers are more powerful, easier to use, and offer an option for styling with Zurb Foundation.

In your **Gemfile**, you've already added:

```
gem 'simple_form'
```

and previously run `$ bundle install`.

Run the generator to install SimpleForm with a Zurb Foundation option:

```
$ rails generate simple_form:install --foundation
```

which installs several configuration files:

```
config/initializers/simple_form.rb
config/initializers/simple_form_foundation.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 Zurb Foundation. 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 Zurb Foundation styling. Thanks to the open source efforts of the Zurb firm, 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 Foundation framework.

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 (just one right now, for "Home").

Here's a troubleshooting tip. If clicking the menu icon doesn't reveal a drop-down menu, the application may not be loading the Foundation JavaScript library. Make sure that the file **app/assets/javascripts/application.js** contains:

```
//= require jquery
//= require jquery_ujs
//= require turbolinks
//= require foundation
//= require_tree .
$(function() {
  $(document).foundation();
});
```

Next we'll add "About" and "Contact" pages to the application.

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**:

```
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 20

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**:

```
<% 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. He founded Acme Manufacturing, later acquired by the Advent
Corporation, to commercialize the product. Mr. Kadigan is now retired and lives in
Middlehampton where he raises Griadium frieda.</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**:

```
<% content_for :title do %>Contact<% end %>
<h3>Contact</h3>
```

This is a placeholder page we'll use to test the navigation link we've already created.

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. If we wanted to add the route explicitly to the file **config/routes.rb**, the file would look like this:

```
LearnRails::Application.routes.draw do
  get "/pages/*id", to: 'pages#show'
  root to: 'visitors#new'
end
```

Again, you don't need to add the code above because the High Voltage gem already provides the route.

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.html.erb** with this:

```
<ul>
  <li><%= link_to 'Home', root_path %></li>
  <li><%= link_to 'About', page_path('about') %></li>
  <li><%= link_to 'Contact', page_path('contact') %></li>
</ul>
```

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," it is likely you copied code into the **config/routes.rb** file that was intended only for explanation. The **config/routes.rb** file should look like this:

```
LearnRails::Application.routes.draw do
  root to: 'visitors#new'
end
```

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 21

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.

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:

```
LearnRails::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 `rake routes` command to see our routes in the console:

```
$ rake 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 `rake routes` command is somewhat cryptic but confirms we've created the routes we need.

The first item in the `rake 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 (ContactsController)
- `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 Zurb Foundation 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**:

```
<% content_for :title do %>Contact<% end %>
<h3>Contact</h3>
<div class="large-6">
  <%= 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: 'button radius' %>
  <% 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="large-6">` uses a Zurb Foundation class to set the width of the form
- `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}"
    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}"
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', contents: '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}"
  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.

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.

Notice what appears in the console log:

```
Started POST "/contact" for 127.0.0.1 at ...
Processing by ContactsController#process_form as HTML
  Parameters: {"utf8"=>"✓", "authenticity_token"=>"rQJfWNurHbEI3RBj/
myfUkcbkeX5cgQ06y4y91Jqthw=", "contact"=>{"name"=>"Daniel Kehoe",
"email"=>"daniel@danielkehoe.com", "content"=>"Looking forward to your birthday!"},
"commit"=>"Submit"}
DEBUG: params are{"utf8"=>"✓", "authenticity_token"=>"rQJfWNurHbEI3RBj/
myfUkcbkeX5cgQ06y4y91Jqthw=", "contact"=>{"name"=>"Daniel Kehoe",
"email"=>"daniel@danielkehoe.com", "content"=>"Looking forward to your birthday!"},
"commit"=>"Submit", "action"=>"process_form", "controller"=>"contacts"}
Redirected to http://localhost:3000/
Completed 302 Found in 0ms (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
- *for 127.0.0.1* – the IP address for localhost
- *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.

Implementing the “Rails Way”

Our initial implementation of the contact form is consistent with the earliest approach to web application development, 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. We'll replace this with something better:

```
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
```

Our "segregation of concern" 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.

ActiveRecord

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 ActiveRecord class interfaces with SimpleForm to provide sophisticated validation and error handling. We can inherit behavior from the ActiveRecord class to add validation and error handling to any model we create.

SimpleForm will recognize ActiveRecord 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 use the activerecord-tableless gem to disable the database features of ActiveRecord.

Note: There's another way to create a model without a database using only the ActiveModel class, described in the RailsCasts: ActiveModel screencast. Either approach is fine; we're using the activerecord-tableless gem because a tableless implementation using ActiveModel requires an understanding of Ruby modules, get and set methods, and object initialization. It's just easier to use the activerecord-tableless gem.

Let's set up a model that inherits from ActiveRecord.

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

```
class Contact < ActiveRecord::Base
  has_no_table

  column :name, :string
  column :email, :string
  column :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
```

We give the model the name "Contact" and inherit from the ActiveRecord class.

We use the `has_no_table` directive from the activerecord-tableless gem to disable database features of ActiveRecord.

We specify attributes (data fields) for the model by using the `column` keyword from the activerecord-tableless gem. These match the fields in the contact form.

ActiveRecord 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. ActiveRecord does all the rest.

Let's change our contact form to use our new model.

Remove the Contact Page

We're implementing a model-view-controller architecture for our Contact feature. That means we have a Contact model, a ContactsController, and we'll need view files in the **app/views/contacts/** folder.

Start by removing the file **app/views/pages/contact.html.erb**:

```
$ rm app/views/pages/contact.html.erb
```

In a minute, we'll replace it with a file in the **app/views/contacts/** folder.

Let's create the **app/views/contacts/** folder:

```
$ mkdir app/views/contacts/
```

Change Navigation Links

Now that we've removed the file **app/views/pages/contact.html.erb**, we'll change the navigation links.

Change the file **app/views/layouts/_navigation.html.erb**:

```
<ul class="nav">
  <li><%= link_to 'Home', root_path %></li>
  <li><%= link_to 'About', page_path('about') %></li>
  <li><%= link_to 'Contact', new_contact_path %></li>
</ul>
```

Our new route `new_contact_path` doesn't yet exist. We'll complete our move to the model-view-controller architecture by adding the appropriate routes.

Next we'll add a new Contact page.

Create a New Contact Page

Earlier, I said 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.

We'll soon create a controller that assigns the Contact model to the `@contact` instance variable. We'll use that as an argument for the SimpleForm view helper: `simple_form_for @contact do |form| ... end`.

We'll also add an `error_notification` method provided by the form object.

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

```
<% content_for :title do %>Contact<% end %>
<h3>Contact</h3>
<div class="large-6">
  <%= 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: 'button radius' %>
  <% 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.

SimpleForm uses the `@contact` instance variable to name the form, set a destination for the form data (the ContactsController#create action), 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 ApplicationController, 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.

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.erb** 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**:

```
class ContactsController < ApplicationController

  def new
    @contact = Contact.new
  end

  def create
    @contact = Contact.new(secure_params)
    if @contact.valid?
      # TODO save data
      # 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 ActiveRecord parent 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 save data (not yet implemented), send a message (not yet implemented), 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.html.erb** 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:

```
LearnRails::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 `rake routes` command to see these in the console:

```
$ rake 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 `rake routes` command shows we've created the routes we need.

We're ready to test the model-view-controller implementation of the Contact feature.

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, "Can't be blank," pointing to the name field.

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 a field is required but blank, SimpleForm will use JavaScript to point to the error before the form is submitted. 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.

Now let's implement code to store the data in a Google Drive spreadsheet.

Chapter 22

Spreadsheet Connection

In the last chapter, we added a contact form to the website. When a visitor submits the form, we display an acknowledgment message. Now we want to capture the data for later analysis or review.

We've purposely chosen not to implement a database application so we can focus on web application basics. Though many Rails applications are backed by databases, a database adds complexity to a Rails application. One of the requirements that adds complexity is authentication and authorization. If data is stored in an application database, we have to implement access control so only an administrator can view it.

Fortunately, Google Drive (formerly known as Google Docs) gives us an easy way to store and access our visitor data without a database. It's an elegant solution. We can use the Google Drive API (application programming interface) to save form data to a spreadsheet that is stored in Google Drive. We don't have to implement authentication and authorization in our application because Google Drive already manages user access. Our application will send the data to a spreadsheet and our imaginary client, Mr. Foobar Kadigan, can access the data on Google Drive.

Like other computer-literate people in business, Mr. Kadigan has experience with spreadsheets. Making data available in a spreadsheet makes it easy for an administrator or a website owner to analyze or review the data.

User Story

Let's plan our work with a user story:

```
*Save Data to a Spreadsheet*
As the owner of the website
I want all contact requests saved in a spreadsheet
In order to analyze the data
```

To implement the user story, let's create a feature that saves the data to Google Drive.

Google Drive Gem

We'll use the google_drive gem to connect to the spreadsheet and save data.

The google_drive gem is a Ruby library to read and write data to spreadsheets in Google Drive. It provides convenient Ruby methods to wrap the Google Spreadsheets API. You can see all the features of the google_drive gem by reviewing the google_drive gem API.

In your **Gemfile**, you've already added:

```
gem 'google_drive'
```

and previously run `$ bundle install`.

The google_drive gem requires a username and password to access your Google Drive account. In the "Configure" chapter, we set this in the **config/application.yml** file.

You may have set up your Google account for 2-step verification, which sends a unique code to your mobile phone each time you log in from an unfamiliar device. If your Google account requires two-factor authentication, you can turn off 2-step verification or create a new Gmail account for use with this tutorial.

Implementation

We will use the API from the google_drive gem to write code that saves data to the spreadsheet. But where should we put the code?

At first glance, it looks like it could be added to the controller. But is there somewhere else to put the code?

When you ask such a question, you are putting on the cape of the software architect. Your decision will be both practical and aesthetic, aligning (hopefully) with tried-and-true software design patterns, and impacting the maintainability of the code. More than any other area of programming, this challenge requires skills honed by experience and informed by reading and discussion with peers. Given a choice of several places to insert the new code, the beginner might say, "Does it really matter?" and make an arbitrary decision. But to the experienced software engineer, the decision is at the heart of the craft.

Let's consider our options.

In this case, the form data is received by the controller. It would be a small extra step to add code to the Contacts controller `create` method to connect to Google Drive to save the visitor's email address and contact message. The code will work, though seasoned Rails developers will raise an eyebrow. Why?

If you think about it, saving data to Google Drive is a data operation, and all data manipulation should be handled by a model.

Rails is opinionated, which means there is often a "Rails way" that is preferred to other approaches. One of the slogans of the "Rails way" is "skinny controller, fat model." The slogan exists to remind developers that separation of concerns makes more modular, maintainable programs. Data manipulation goes in a model. Controllers should contain only enough code to instantiate a model and render a web page.

Consequently, we'll add a method to the Contacts model named `update_spreadsheet` that will use the google_drive gem to connect to Google Drive and save the data. We'll call the `@contact.update_spreadsheet` method from the Contacts controller `create` method.

Modify the Contact Model

We'll add our spreadsheet code to the Contact model.

Replace the contents of the file **app/models/contact.rb**:

```
class Contact < ActiveRecord::Base
  has_no_table

  column :name, :string
  column :email, :string
  column :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

  def update_spreadsheet
    connection = GoogleDrive.login(ENV["GMAIL_USERNAME"], ENV["GMAIL_PASSWORD"])
    ss = connection.spreadsheet_by_title('Learn-Rails-Example')
    if ss.nil?
      ss = connection.create_spreadsheet('Learn-Rails-Example')
    end
    ws = ss.worksheets[0]
    last_row = 1 + ws.num_rows
    ws[last_row, 1] = Time.new
    ws[last_row, 2] = self.name
    ws[last_row, 3] = self.email
    ws[last_row, 4] = self.content
    ws.save
  end

end
```

We'll call the new `update_spreadsheet` method from the controller.

The google_drive gem gives us a `GoogleDrive` class.

Create a connection to Google Drive by passing your credentials to the `login` method. Here's where we use the environment variables we set in the **config/application.yml** file using the figaro gem.

We look for a spreadsheet named "Learn-Rails-Example." The first time we attempt to save data, the spreadsheet will not exist, so we use the `create_spreadsheet` method to create it. If it already exists, the `spreadsheet_by_title` method will find it.

A single spreadsheet file can contain multiple *worksheets*. We'll use only one worksheet to store our data, designated as "worksheet 0" (we count from zero).

Here the code gets a little tricky. You might expect the API to provide an "append row" method. In fact, we have to retrieve a count of rows, and then add one, to calculate the row number of the last empty row.

We add data on a cell-by-cell basis, by designating the row number and column number of a cell. We add the current date and time using the Ruby API method `Time.new` to the first cell in the last row. Then we add `name`, `email`, and `content` attributes to additional columns (we refer to the current instance of the class by using the keyword "self").

Setting the cell value doesn't save the data. We explicity call the worksheet `save` method to update the worksheet.

Modify the Contacts Controller

Our Contact model now has a method to save data to a spreadsheet.

We'll update the Contacts controller to save the data.

Replace the contents of the file **app/controllers/contacts_controller.rb**:

```
class ContactsController < ApplicationController

  def new
    @contact = Contact.new
  end

  def create
    @contact = Contact.new(secure_params)
    if @contact.valid?
      @contact.update_spreadsheet
      # 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 added only one line, the `@contact.update_spreadsheet` statement.

When the visitor submits the form, the ContactsController#create action is called. The `create` method will instantiate a new Contact model using the data from the form after laundering

the parameters. If the validation check succeeds, we save data to the spreadsheet, set a flash notice, and redisplay the home page.

In only a few lines of code, we've added data storage using Google Drive.

Test the Application

Make sure the web server is running:

```
$ rails server
```

Open a web browser window and navigate to http://localhost:3000/.

Click the "Contact" link and try submitting the form. You'll see an acknowledgment message.

You'll get an error "Error=BadAuthentication Info=InvalidSecondFactor" if your Google account is set for 2-step verification. Change your Google account settings to turn off two-factor authentication if you want to complete the test successfully.

Visit your Google Drive account ("Drive" is in the navigation bar when you visit the Google Search or Gmail home pages). You'll see a list of Google Drive files. The newest one will be a **Learn-Rails-Example** spreadsheet. Open the file and you will see the data from the contact form. Whenever a visitor submits the contact form, the spreadsheet will update within seconds.

Git

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

```
$ git add -A
$ git commit -m "save data to a spreadsheet"
$ git push
```

We've got a fully functional contact form that stores data in a Google Drive spreadsheet.

Now let's add the code to email the form data to the site owner.

Chapter 23

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 one file and one folder:

- **app/mailers/user_mailer.rb**
- **app/views/user_mailer**

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.rb**:

```
class UserMailer < ActionMailer::Base
  default from: "do-not-reply@example.com"

  def contact_email(contact)
    @contact = contact
    mail(to: ENV["OWNER_EMAIL"], from: @contact.email, :subject => "Website Contact")
  end
end
```

The `UserMailer` class inherits behavior from the `ActionMailer` 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 an environment variable for your email address in the file **config/application.yml**. If you haven't done so, do it now. By inserting the environment 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
```

Replace the contents of the file **app/controllers/contacts_controller.rb**:

```
class ContactsController < ApplicationController

  def new
    @contact = Contact.new
  end

  def create
    @contact = Contact.new(secure_params)
    if @contact.valid?
      @contact.update_spreadsheet
      UserMailer.contact_email(@contact).deliver
      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` 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.

You may see a warning message when you log into your Gmail account, indicating that someone used your credentials to send email. You can dismiss the warning as you know it was yourself.

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 Gmail 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 Gmail 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.

Unfortunately, asynchronous mailing, which requires a *queueing system*, is an advanced topic for Rails developers.

Earlier I wrote that Rails, as a framework, is not complete. This is an example. The developer community has explored the possibility of implementing a standard queueing system for Rails. In fact, an early version of Rails 4.0 contained a queueing system but it was dropped because it did not fully address several complicated issues (see What happened to the Rails 4 Queue API?).

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 may need to learn how to implement a queueing system. You'll find examples in more advanced tutorials.

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 24

Mailing List

Even as other messaging avenues become increasingly popular, such as SMS text 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 saving data to Google Drive, or connecting to Gmail 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**:

```
<% content_for :title do %>Foobar Kadigan<% end %>
<% content_for :description do %>Website of Foobar Kadigan<% end %>
<img src="http://lorempixel.com/1170/600/cats/1">
<div style="text-align: center; margin-top: 20px">
  <div class="row">
    <div class="large-6 columns">
      <h2>Stay in touch.</h2>
    </div>
    <div class="large-6 columns">
      <div class="large-6">
        <%= simple_form_for @visitor do |f| %>
          <%= f.error_notification %>
          <%= f.input :email, label: false, :placeholder => 'Your email address...' %>
          <br/>
          <%= f.submit "Sign up for the newsletter", :class => "button radius" %>
        <% end %>
      </div>
    </div>
  </div>
</div>
```

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.

Much of the markup is divs to create layout. The first div creates a container with centered text, offset from the photo by 20 pixels. If we were being strict about separation of concerns, we'd specify this rule in a CSS stylesheet instead of using a `style` attribute.

The next two divs apply a grid from Zurb Foundation 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 button will contain the text, "Sign up for the newsletter," and we apply Zurb Foundation button styling.

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**:

```
class Visitor < ActiveRecord::Base
  has_no_table
  column :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::API.new
    result = mailchimp.lists.subscribe({
      :id => ENV['MAILCHIMP_LIST_ID'],
      :email => {:email => self.email},
      :double_optin => false,
      :update_existing => true,
      :send_welcome => true
    })
    Rails.logger.info("Subscribed #{self.email} to MailChimp") if result
  end

end
```

Once again, we inherit behavior from the ActiveRecord parent class and use the `has_no_table` keyword from the activerecord-tableless gem to disable ActiveRecord's database functionality.

We create the email attribute and set validation requirements.

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. The Gibbon gem looks in the environment variables for the `MAILCHIMP_API_KEY` value so we don't need to specify it here. We assign the Gibbon object to the `mailchimp` variable (we could name it anything).

Gibbon offers a `lists.subscribe` method which takes five parameters:

- id – environment variable to identify the MailChimp list
- email – address of the visitor (inside a hash)
- double_optin – setting `true` sends a double opt-in confirmation message
- update_existing – updates a subscriber record if it already exists
- send_welcome – sends a "Welcome Email" to the new subscriber

The parameters are described further in the MailChimp API Documentation.

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**:

```
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:

```
LearnRails::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 `rake routes` command to see these in the console:

```
$ rake 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 `rake 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 mailing list 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 menu and click "Signup forms." Then click "Link to a form." On the "Create Forms" page, there is a dropdown 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 25

Deploy

You've been running the default Ruby WEBrick 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, WEBrick 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, EngineYard, OpenShift, Cloud Foundry, or Shelly Cloud.

You may already be using a shared web hosting service such as GoDaddy or DreamHost for a static website or WordPress site. Be skeptical if a shared web hosting service 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. A PaaS platform provides 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.

Our Rails Heroku Tutorial goes into more detail.

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). Heroku provides one dyno for every web application for free. For personal projects, you can run your Rails application on a single dyno and never incur a charge.

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 to respond to every request without delay, you can run two dynos. Heroku charges $35 per month for a second dyno running full time (a dyno is billed at $0.05/hour).

A single dyno can serve thousands of requests per second, but performance depends greatly on your application. With the Ruby WEBrick server, Rails processes only one request at a time. Heroku doesn't support WEBrick, but as a default it supports Thin, a similar "single-threaded, non-concurrent" web server. Serving a typical Rails application that takes 100ms on average to process each request, Thin can accommodate about 10 requests per second per dyno, which is adequate for a personal project.

If traffic surges on your website and exceeds 10 requests per second, you can scale up. First, you can replace the default Thin web server with the Unicorn web server which handles *concurrent* requests. Configuring Unicorn requires more expertise than Thin, but Heroku recommends it. Second, you can double the size of Heroku's dynos to handle more requests. Finally, you can buy more dynos, adding as many dynos as you need to handle traffic. This is where convenience comes at a price. You won't need system administration expertise to deploy a website on Heroku but you'll pay a premium to host a high-traffic site.

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 gems that Heroku needs:

- pg – PostgreSQL gem
- thin – web server
- rails_12factor – logging and static assets

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.

The Thin web server is easy to use and requires no configuration. Note that Heroku recommends Unicorn for handling higher levels of traffic efficiently. Unicorn can be difficult to setup and configure, so we're using Thin for our tutorial application.

On Heroku, Rails 4.0 needs an extra gem to handle logging and serve CSS and JavaScript assets. The rails_12factor gem provides these services.

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

Gemfile

```
source 'https://rubygems.org'
ruby '2.0.0'
gem 'rails', '4.0.0'
gem 'sass-rails', '~> 4.0.0'
gem 'uglifier', '>= 1.3.0'
gem 'coffee-rails', '~> 4.0.0'
gem 'jquery-rails'
gem 'turbolinks'
gem 'jbuilder', '~> 1.2'
gem 'activerecord-tableless'
gem 'compass-rails', '~> 2.0.alpha.0'
gem 'figaro'
gem 'gibbon'
gem 'google_drive'
gem 'high_voltage'
gem 'simple_form'
gem 'zurb-foundation'
group :development do
  gem 'better_errors'
  gem 'quiet_assets'
  gem 'rails_layout'
  gem 'sqlite3'
end
group :production do
  gem 'pg'
  gem 'rails_12factor'
  gem 'thin'
end
```

We have to run `bundle install` because we've changed the Gemfile. The gems we've added are only needed in production so we don't install them on our local machine. When we deploy, Heroku will read the Gemfile and install the gems in the production environment. We'll run `bundle install` with the `--without production` argument so we don't install the new gems locally:

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

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

```
$ git add -A
$ git commit -m "gems for Heroku"
$ git push
```

Precompile Assets

In development mode, the Rails asset pipeline "live compiles" all CSS and JavaScript files and makes them available for use. Compiling assets adds processing overhead. In production, a web application would be slowed unneccesarily if assets were compiled for every web request. Consequently, we must precompile assets before we deploy our application to production.

When you precompile assets for production, 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/application.css.scss**. You must commit the compiled files to your git repository before deploying.

Here's how to precompile assets and commit to the Git repo:

```
$ RAILS_ENV=production rake assets:precompile
$ git add -A
$ git commit -m "assets compiled for Heroku"
$ git push
```

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.

If you don't precompile assets for production, all web pages will look strange. They won't have CSS styling.

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:

```
# public/robots.txt
# To allow spiders to visit the entire site comment out 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/signup/devcenter 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.

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.

Set a Domain Name

Earlier, when we created the **config/application.yml** file and set environment variables, we left a placeholder for a domain name in the file. Now we can replace the placeholder.

Open the file **config/application.yml**:

```
# Add account credentials and API keys here.
# See http://railsapps.github.io/rails-environment-variables.html
# This file should be listed in .gitignore to keep your settings secret!
# Each entry sets a local environment variable and overrides ENV variables in the Unix
shell.
GMAIL_USERNAME: Your_Username
GMAIL_PASSWORD: Your_Password
MAILCHIMP_API_KEY: Your_MailChimp_API_Key
MAILCHIMP_LIST_ID: Your_List_ID
DOMAIN_NAME: example.com
OWNER_EMAIL: me@example.com
```

If you don't have a custom domain name, use the domain name you've chosen for deployment on Heroku. Replace `example.com` with `myapp.herokuapp.com`, replacing `myapp` with the name that Heroku has accepted for your application.

If you already have a custom domain name in the file, you don't have to change anything, but you will have to set up Heroku to use your custom domain name. That involves setting up DNS, which we won't cover in this tutorial.

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 Gmail from Heroku, add the following to your **config/environments/production.rb** file:

```
config.action_mailer.default_url_options = { :host => ENV["DOMAIN_NAME"] }
config.action_mailer.delivery_method = :smtp
config.action_mailer.perform_deliveries = true
config.action_mailer.raise_delivery_errors = false
config.action_mailer.default :charset => "utf-8"
config.action_mailer.smtp_settings = {
  address: "smtp.gmail.com",
  port: 587,
  domain: ENV["DOMAIN_NAME"],
  authentication: "plain",
  enable_starttls_auto: true,
  user_name: ENV["GMAIL_USERNAME"],
  password: ENV["GMAIL_PASSWORD"]
}
```

You'll need to specify the unique name you've selected for your hosted application. We're using an environment variable `ENV["DOMAIN_NAME"]` in two places in the file. Be sure you set the environment variable for the domain name in the file **config/application.yml** in the previous step.

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!

Next we'll set Heroku environment variables.

Set Heroku Environment Variables

You'll need to set the configuration values from the **config/application.yml** file as Heroku environment variables.

With the figaro gem, just run:

```
$ rake figaro:heroku
```

Alternatively, you can set Heroku environment variables directly.

Here's how to set environment variables directly on Heroku with `heroku config:add`.

```
$ heroku config:add GMAIL_USERNAME='myname@gmail.com' GMAIL_PASSWORD='secret'
$ heroku config:add MAILCHIMP_API_KEY='mykey' MAILCHIMP_LIST_ID='mylistid'
$ heroku config:add OWNER_EMAIL='me@example.com' DOMAIN_NAME='myapp.herokuapp.com'
```

You can 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 you've run `RAILS_ENV=production rake assets:precompile`. Run it each time you change your CSS or JavaScript files.

Be sure to commit your code to the Git local repository before you push to Heroku:

```
$ git add -A
$ git commit -m "assets compiled for Heroku"
$ git push
```

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
```

The push to Heroku takes several minutes. You'll see a sequence of diagnostic messages in the console, beginning with:

```
-----> Ruby/Rails app detected
```

and finishing with:

```
-----> Launching... 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.

If you've changed anything in the **assets** folder (including images, JavaScript, or stylesheets), you'll need to precompile assets. A typical update scenario looks like this:

```
$ git add -A
$ git commit -m "revised application"
$ RAILS_ENV=production rake assets:precompile
$ git add -A
$ git commit -m "assets compiled for Heroku"
$ git push
$ git push heroku master
```

Visit Your Site

Open your Heroku site in your default web browser:

```
$ heroku open
```

Your application will be running at http://my-app-name.herokuapp.com/.

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. Your issue may have been encountered and addressed by others.

You can also check the Heroku Dev Center or the Heroku Google Group.

Chapter 26

Analytics

In earlier chapters, we've built the tutorial application and deployed it for hosting on Heroku.

We've left something out. Though not obvious, it's very important.

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 4.0. The article on Analytics for Rails looks at various approaches and explains how Google Analytics works.

For this tutorial, we'll use the Segment.io service. The service provides an API to send analytics data to dozens of different services, including Google Analytics.

Segment.io

Segment.io 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- and medium- volume websites, providing one million API calls (page views or events) per month at no cost. There is no charge to sign up for the service.

Using Segment.io 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.io 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.io. Sign up for Segment.io.

You will need accounts with each of the services that you'll use via Segment.io.

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-XXXXXXX-XX`. You won't need the tracking code snippet as we will use the Segment.io JavaScript snippet instead.

You'll check your Google Analytics account later to verify that Google is collecting data.

Installing the JavaScript Library

Segment.io 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.io library loads asynchronously, so it won't affect page load speed.

The Segment.io 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.io JavaScript snippet to a file named **app/assets/javascripts/segmentio.js**. The manifest directive `//= require_tree .` in the file **app/assets/javascripts/application.js** 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/segmentio.js** file.

Create a file **app/assets/javascripts/segmentio.js** and include the following:

```
var analytics=analytics||[];(function(){var
e=["identify","track","trackLink","trackForm","trackClick","trackSubmit","page",
"pageview","ab","alias","ready","group"],t=function(e){return
function(){analytics.push([e].concat(Array.prototype.slice.call(arguments,0)))}};for(var
n=0;n<e.length;n++)analytics[e[n]]=t(e[n])})(),analytics.load=function(e){var
t=document.createElement("script");t.type="text/javascript",t.async=!0,
t.src=("https:"===document.location.protocol?"https://":"http://")+
"d2dq2ahtl5zl1z.cloudfront.net/analytics.js/v1/"+e+"/analytics.min.js";var
n=document.getElementsByTagName("script")[0];n.parentNode.insertBefore(t,n)};
analytics.load("YOUR_API_TOKEN");
$(document).on('page:load', function() {
  console.log('page loaded');
  analytics.pageview();
  analytics.trackForm($('#new_visitor'), 'Signed Up');
  analytics.trackForm($('#new_contact'), 'Contact Request');
})
```

If you find you can't copy this code from this page, you can get it directly from the reference implementation of the tutorial application. The app/assets/javascripts/segmentio.js file is on GitHub.

The Segment.io website offers a minified version of the snippet for faster page loads. We've used it here for convenience. You can look at the non-minified version on the Segment.io website if you want to read the code and comments.

You **must replace** `YOUR_API_TOKEN` with your Segment.io API token. You can find the API token on your "Settings" page when you log in to Segment.io (it is labelled "Your API Key").

We've added extra code to the minified Segment.io JavaScript snippet. The extra code accomodates page view and event tracking, which we'll look at next.

Page View Tracking with Turbolinks

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.io 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 get tracking data with Turbolinks.

To make sure every page is tracked when Rails 4.0 Turbolinks is used, we've already appended the following JavaScript to the **app/assets/javascripts/segmentio.js** file:

```
$(document).on('page:load', function() {
  analytics.pageview();
});
```

This code follows the `analytics.load('YOUR_API_TOKEN');` statement at the end of the file.

Turbolinks fires a `page:change` event when a page has been replaced. The code listens for the `page:load` event and calls the Segment.io `pageview()` method.

Event Tracking

Segment.io 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.io JavaScript library gives us two methods to track events:

- `trackLink`
- `trackForm`

Link tracking can be used to send data to Segment.io 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/segmentio.js** file:

```
$(document).on('page:load', function() {
  analytics.trackForm($('#new_visitor'), 'Signed Up');
  analytics.trackForm($('#new_contact'), 'Contact Request');
})
```

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.

With Google Analytics enabled as a Segment.io integration, you'll see form submissions appear in the Google Analytics Real-Time report, under the "Events" heading.

You can read more about the Segment.io JavaScript library in the Segment.io documentation.

Segment.io Integrations

After installing the Segment.io JavaScript snippet in your application, visit the Segment.io integrations page to select the services that will receive your data. When you log in to Segment.io you will see a link to "Integrations" in the navigation bar.

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-XXXXXXX-XX`. Check the box to "Enable Client-Side Universal Analytics." Accept the other defaults.

Click "Dashboard" in the navigation bar so you can monitor data sent to Segment.io from your application.

Deploy

When you are ready to deploy to Heroku, you must recompile assets and commit to the Git repo:

```
$ git add -A
$ git commit -m "analytics"
$ RAILS_ENV=production rake assets:precompile
$ git add -A
$ git commit -m "assets compiled for Heroku"
$ 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.io in the Segment.io dashboard.

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.io 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.io also integrates with many services which provide conversion tracking.

Chapter 27

Rails Challenges

Rails is popular. Rails is powerful. But Rails isn't easy to learn.

You may have heard of a psychological phenomenon called "resistance." When we struggle with something new, or must adapt to the unfamiliar, we resist. We get discouraged. We complain. Sometimes we feel we should quit.

This chapter is here to help with your resistance.

Its purpose is to acknowledge that, yes, *Rails can be difficult*.

Tens of thousands of people are successfully using Rails. I'll hazard a guess that none are significantly smarter, more motivated, or a better student than you. Perhaps some of them had more time to study or better access to mentors, but these factors simply accelerate the speed of learning Rails. If you get discouraged, or think Rails is too hard, recognize that you are encountering your own resistance, not any genuine limitation. Take a break, set aside your learning materials, and come back when your natural curiosity and eagerness has returned.

Sometimes resistance attaches to imaginary problems (like "I'm not smart enough"). Just as often, resistance attaches to real problems, but magnifies them into insurmountable obstacles ("Rails is impossible to use on Windows!"). The best way to overcome these obstacles is to acknowledge the resistance, investigate the obstacle, and seek support from peers.

This chapter describes some of things that make Rails difficult.

These Rails challenges are obstacles, but other people overcame them. You can, too.

The list is incomplete. If you've encountered a Rails challenge that isn't listed here, email me at daniel@danielkehoe.com and I will add your suggestion to the next revision of the book.

It is difficult to install Ruby.

The installation process for Ruby on Rails is more difficult than downloading and installing any consumer software applications. You are setting up a development environment and you need system software as well as Ruby. Depending on what you've done before, you may have altered your system, introducing potentials for conflicts. This book provides links to good installation guides in the "Get Started" chapter. But installation instructions can't accommodate the specific configuration of your computer. Sometimes you just have to look for someone to help. I've also suggested using Nitrous.io, a hosted development environment.

Rails is a nightmare on Windows.

Windows is very popular, so why is it difficult to develop with Rails on Windows? It seems the Rails community has a bias against Windows. It does, and there's a reason. Rails is an open source project. Most open source developers use Unix-based system tools. It is difficult and time-consuming to convert Unix-based system tools to the Microsoft Windows operating system. Open source developers prefer to spend their time maintaining and improving their Unix-based projects. And expert Windows developers are seldom interested in porting Unix-based system tools to Windows. So system utilities such as RVM are not available for Windows. And developers who create gems are seldom interested in spending time to solve the problems that arise when code has to be adapted for the idiosyncrasies of the Windows platform. This situation is not going to change, so you have to make a choice. Stay with Windows or get comfortable with Unix-based systems.

Why do I have to learn Git? It is difficult.

Real software development requires version control and Git is the standard tool for Rails developers. If all you do is build applications as a classroom exercise, you don't need to learn Git. You can skip all the parts of the book that mention Git. But sooner or later, if you start doing real projects, you'll need Git. Simple Git commands are not difficult to learn. When you've developed your skills and confidence you can learn the more advanced Git functions, such as branching.

RVM seems unnecessary. Why worry about versions?

For simple projects you don't need RVM. This book introduces RVM and prepares you to handle version conflicts. As you tackle more complex projects, and as new versions of Rails are released, you'll face version issues and RVM will be helpful.

Do I really need to learn about testing?

For student projects, no, you don't need to learn about testing. But as soon as money or reputation is at stake on a project, you'll need to begin using test-driven development. This book doesn't teach TDD because it is overwhelming for a beginner. After you complete the book, intermediate-level tutorials will introduce you to testing. Once you've grasped the basics of Rails, testing will become easy, and it actually is fun and satisfying.

Rails error reporting is cryptic.

Actually, Rails error reporting is quite good. Stack traces are detailed and error messages are descriptive. Beginners have a problem because the stack traces and error messages provide a technical analysis of a problem in terms that an experienced developer can understand. If error reporting was "simplified" it might not be as intimidating but it would not as accurate. It's up to you to gain enough knowledge to understand the error messages. Finally, the error

reporting mechanism can point you to the line in your code that triggers a problem, but it can't know what you trying to do, or describe the error in anything but technical terms.

There is too much magic.

The Rails "convention over configuration" principle leads to obscurity. Default behavior often looks like magic because the underlying implementation is hidden in the depths of the Rails code library. If you like to know how things work, this can be frustrating. You really have only two choices when you encounter Rails magic. You can take time to dig into the source code. If you do so, you'll encounter frustration as you encounter complex and sophisticated code, but you may also improve your understanding and skill as a Ruby programmer. Or you can take on faith that "it just works." Often, you just need to use the convention several times in different projects to get comfortable with the magic and stop worrying that you don't fully understand it.

Rails contains lots of things I don't understand.

If you look at the Rails directory structure, you'll see many files and folders. If you look at the Rails API, or pick up a Ruby tutorial, you'll also see code that is unfamiliar. This book has described some of what you see. As you build more applications, you will gain proficiency and master more of Rails and Ruby. Yet even as you gain mastery of Rails, there will be aspects that remain unfamiliar. Don't let the sheer complexity stop you. The truth is, you don't have to know "all" of Rails or Ruby to build web applications.

There is too much to learn.

Very true. To be a full-stack web developer you need to know HTML, JavaScript, CSS, Ruby, testing, databases, and much, much more. You might think that developers who started ten years ago have an advantage because there wasn't as much to learn when they started. But today there are many more high-quality tutorials and educational programs to accelerate your learning. And resources like Google and Stack Overflow have many more answers to questions. As the knowledge domain has grown, so have the learning resources. You don't have to learn everything. Get a foundation in the basics and then dive deep as a specialist in an area that appeals to you.

It is difficult to find up-to-date advice.

Rails has been around since 2004 with major new versions released every two years. Chances are, answers to questions you find on Stack Overflow or Google were written for an older version of Rails. There is no easy way to determine if the answer is out of date. A particular aspect of Rails may have changed—or not. Even worse, the answer may work, but there may be a better way that reflects current best practices. To filter the outdated in Google, use the "Search Tools" options for specifying a timeframe. Look closely at the date of a blog posting or Stack Overflow answer. Try to find a newer answer. Usually, if there are a series of

answers and things have changed, you'll see the current best answer. If you're uncertain, don't be shy about posting your question to Stack Overflow. More importantly, make it your business to keep up with the community, reading Peter Cooper's Ruby Weekly email newsletter or his daily RubyFlow site.

It is difficult to know what gems to use.

There are so many gems available for Rails. Some add useful features, like tagging or a mailing list API. Some are basic, such as gems for a database or front-end framework. Even among basic gems, Rails offers choices. Which are best? The Ruby Toolbox can help, but mostly you will find guidance from looking at example projects and noticing what other developers are using. There's wisdom in the crowd.

Rails changes too often.

If you look at the Ruby on Rails Release History you'll see there is a new major release approximately every 1.5 years. Each major release is well tested and relatively free of bugs. But new features or new approaches often require rewrites of older applications. Commercial software products often make a priority of keeping the API consistent over time. That's not Rails. Rails is an open source project and the core team embraces innovation. The maintainers expect that you'll keep up with changes.

It is difficult to transition from tutorials to building real applications.

Copying and pasting from tutorials is a good way to begin learning Rails. But you'll only become a skilled Rails developer when you build something that is not shown in a tutorial. The first few hours (or days) when you start building a custom application can be very difficult. Focus on the basics that are described in this book. Start with user stories. Build pieces that you know how to do. Look for code samples on blogs or GitHub or Stack Overflow. Try "spikes," little experiments that test ideas for implementation. Seek advice from peers or mentors. At first it will be slow going. But you will pick up momentum.

I'm not sure where the code goes.

If you follow tutorials, you'll learn "where the code goes" with the model–view–controller design pattern. With a sense of the request-response cycle, RESTful actions in the controller, and a few guidelines such as "skinny controller, fat model" you'll be able to build intermediate-level Rails applications. Front-end code, particularly JavaScript, can be difficult because not a lot has been published about Rails best practices. In particular, the Rails asset pipeline can be confusing for anyone who has done front-end development without Rails. If you don't know what you're supposed to do, do whatever works, then look for someone who can help you by providing a code review.

People like me don't go into programming.

Until recently, most Rails developers have been young men with an engineering background. For people who don't fit the stereotypical profile, it can be hard to find role models or peers who demonstrate that Rails is something everyone can learn. The challenge can be subtle, as when you have the feeling that maybe if you were different you'd find it easier to make progress. Or the challenge can be overt, when behavior of fellow students or co-workers is disturbing or hurtful (often they don't even know!). Lack of diversity, and the cluelessness that accompanies it, is unfortunate in the Rails community. But many people are working to make the community more welcoming and inclusive. Organizations such as Rails Girls and Railsbridge are creating more diversity in the community. You may find support from peers there to affirm that you, too, are entitled to knowledge and success.

Chapter 28

Credits and Comments

Credits

Daniel Kehoe wrote the book and implemented the application.

The book was created with the encouragement, financial support, and editorial assistance of hundreds of people in the Rails community.

Financial Backers

The following individuals and organizations provided financial contributions of over $50 to help launch the book. Please join me in thanking them for their encouragement and support.

Al Zimmerman
Avi Flombaum
Brian Hays
Charles Treece
Dave Doolin
Denzil Villarico
Derek Rockwell
Frank Castle
Gerard de Brieder
GoodWorksOnEarth.org
Hanspeter Leupin
Harald Lazardzig
Gant Laborde
Jared Koumentis
Jeff Whitmire
Jesse House
Joost Baaij
Kathleen Sidenblad
Logan Hasson
Matt Esterly
Mike Gilbert
Paul Philippov
Robert Nadar
Rogier Hof
Ross Kinney
Susan Wilson

Sven Fuchs
Tom Michel
Youn Shin Kang

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, Ben Swee, Brandon Schabel, 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, Joanne Daudier, Joel Dezenzio, Jonah Ruiz, Jonathan Lai, Jonathan Miller, Jordan Stone, 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, Silvia Obajdin, Stas Suşcov, Stefan Streichsbier, Sven Fuchs, Tam Eastley, Tim Goshinski, Timothy Jones, Tom Connolly, 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 (CC BY-SA). 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 white cat by Tomi Tapio appears in the screenshot in the Introduction chapter and on the tutorial cover page.

Did You Like the Tutorial?

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.

Any issues with the tutorial application? Please create an issue on GitHub. Reporting (and patching!) issues helps everyone.

Comments

I regularly update the book. Your comments and suggestions for improvements are welcome.

Feel free to email me directly at daniel@danielkehoe.com.